

# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 1

October 4th, 2022

- **Scope**

- Learn the basics of image processing and recognition using deep learning

- **Contents**

- 1 Introduction. Linear models
- 2 Multilayer perceptrons
- 3 Convolutional neural networks
- 4 Recurrent neural networks
- 5 Attention mechanisms
- 6 Computer vision

- **Grading**

- written exam (open book)
- 0.2 points per presence
- 50% of final grade

- **Scope**

- Implement the algorithms presented at the course in PyTorch, using Google Colaboratory

- **Contents**

- 1 Introduction to Python
- 2 Introduction to PyTorch
- 3 Multilayer perceptrons
- 4 Convolutional neural networks
- 5 Recurrent neural networks
- 6 Attention mechanisms
- 7 Computer vision

- **Grading**

- 2 practical tests (open book)
- 50% of final grade

# 1 Introduction. Linear models

## 1.1 Artificial intelligence, machine learning, and deep learning

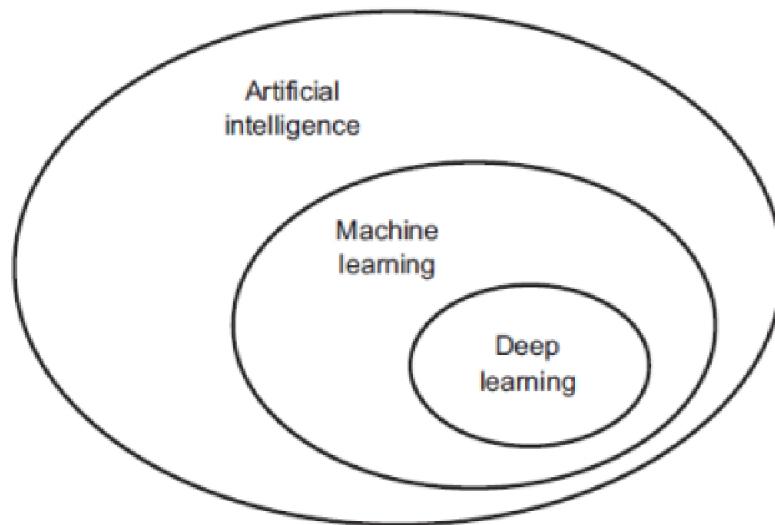


Figure 1: Artificial intelligence, machine learning, and deep learning.

## 1.1.1 Artificial intelligence

- *the effort to automate intellectual tasks normally performed by humans*
- *symbolic AI*: programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge
- peak popularity during the *expert systems* boom of the 1980s
- symbolic AI – suitable to solve well-defined, logical problems, but intractable to figure out explicit rules for solving more complex, fuzzy problems

## 1.1.2 Machine learning

- rather than programmers crafting data processing rules by hand, could a computer automatically learn these rules by looking at data?

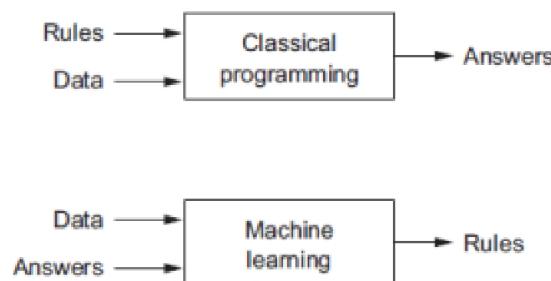


Figure 2: Machine learning: a new programming paradigm.

- a machine learning system is *trained* rather than explicitly programmed
- although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI

## 1.1.3 Learning representations from data

- machine learning discovers rules to execute a data processing task, given examples of what's expected
- so, to do machine learning, we need three things:
  - *Input data points*
  - *Examples of the expected output*
  - *A way to measure whether the algorithm is doing a good job*
- the central problem in machine learning and deep learning is to *meaningfully transform data*: in other words, to learn useful *representations* of the input data at hand – representations that get us closer to the expected output
- representation – a different way to look at data – to *represent* or *encode* data

### 1.1.3 Learning representations from data

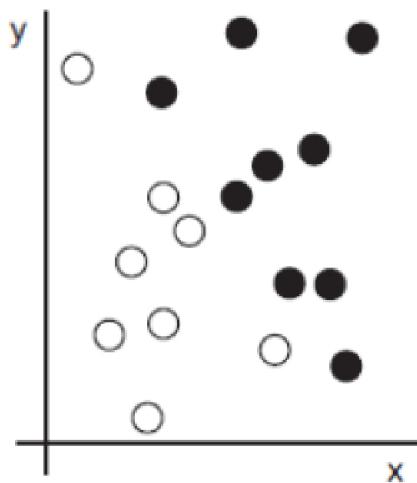


Figure 3: Some sample data.

- inputs?
- expected outputs?
- a way to measure whether the algorithm is doing a good job?

### 1.1.3 Learning representations from data

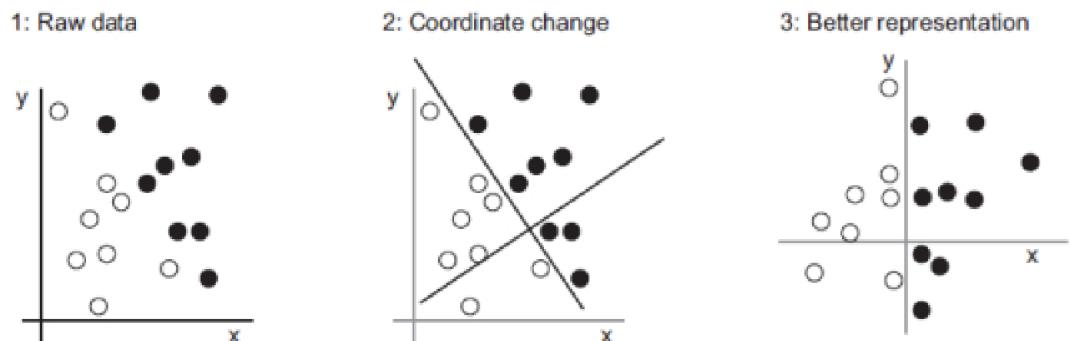


Figure 4: Coordinate change.

- with this representation, the black/white classification problem can be expressed as a simple rule: “Black points are such that  $x > 0$ ”, or “White points are such that  $x < 0$ ”

## 1.1.3 Learning representations from data

- if we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified – machine learning
- *learning*, in the context of machine learning, describes an automatic search process for better representations
- machine learning algorithms aren't creative in finding transformations; they're merely searching through a predefined set of operations, called a *hypothesis space*
- two main types of problems:
  - *classification*, where the goal is to predict a single discrete label for an input data point
  - *regression*, which consists of predicting a continuous value instead of a discrete label

## 1.1.4 Four branches of machine learning

### Supervised learning

- it consists of learning to map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans)
- generally, almost all applications of machine learning that are in the spotlight these days belong in this category, such as speech recognition, image classification, and language translation
- although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following:
  - *Sequence generation*—Given a picture, predict a caption describing it.
  - *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
  - *Object detection*—Given a picture, draw a bounding box around certain objects inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding box coordinates are predicted via vector regression.
  - *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.

# Unsupervised learning

- this branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand
- unsupervised learning is at the core of data analytics, and it's often a necessary step in better understanding a dataset before attempting to solve a supervised learning problem
- *dimensionality reduction* and *clustering* are well-known categories of unsupervised learning

- this is a specific instance of supervised learning, but it's different enough that it deserves its own category
- self-supervised learning is supervised learning without human-annotated labels
- there are still labels involved (because the learning has to be supervised by something), but they're generated from the input data, typically using a heuristic algorithm
- for instance, *autoencoders* are a well-known instance of self-supervised learning, where the generated targets are the input, unmodified
- in the same way, trying to predict the next frame in a video, given past frames, or the next word in a text, given previous words, are instances of self-supervised learning (*temporally supervised learning*, in this case: supervision comes from future input data)

# Reinforcement learning

- long overlooked, this branch of machine learning recently started to get a lot of attention after Google DeepMind successfully applied it to learning to play Atari games (and, later, learning to play Go at the highest level)
- in reinforcement learning, an *agent* receives information about its environment and learns to choose actions that will maximize some reward
- for instance, a neural network that “looks” at a video game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning
- currently, reinforcement learning is mostly a research area and hasn’t yet had significant practical successes beyond games
- in time, however, we expect to see reinforcement learning take over an increasingly large range of real-world applications: self-driving cars, robotics, resource management, education, and so on

## 1.2 Linear regression

- *linear regression* is a very widely used method for predicting a real-valued output (dependent variable, also called the *label* or *target*)  $y \in \mathbb{R}$ , given a set of real-valued inputs (independent variables, also called *features* or *covariates*)  $x_1, x_2, \dots, x_d$
- we assume that the relationship between the independent variables  $x_1, x_2, \dots, x_d$  and the dependent variable  $y$  is *linear*, i.e., that  $y$  can be expressed as a weighted sum of the features  $x_1, x_2, \dots, x_d$ , given some noise on the observations
- also, we assume that the noise follows a normal distribution
- to develop a model for predicting  $y \in \mathbb{R}$ , we need to obtain a *dataset* consisting of known outputs for corresponding inputs
- the dataset is called a *training dataset* or *training set*, and each row is called an *example* or *data point* or *sample*

## 1.2 Linear regression

- we will use  $n$  to denote the number of examples in our dataset
- we index the data examples by  $i$ , denoting each input as  $x_1^{(i)}, x_2^{(i)}, \dots, x_d^{(i)}$  and the corresponding label as  $y^{(i)}$
- the linear regression model is expressed as:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d, \quad (1)$$

where

- $\hat{y}$  is the *predicted value*
- $x_j$  is the  $j$ th *feature* value
- $d$  is the number of features
- $w_j$  is the  $j$ th model *weight* (including the *bias* or *intercept* term  $w_0$  and the *feature weights*  $w_1, w_2, \dots, w_d$ )

## 1.2 Linear regression

- strictly speaking, (1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via weighted sum, combined with a *translation* via the added bias
- the weights determine the influence of each feature on our prediction and the bias just says what the predicted value should be when all of the features take value 0
- given a dataset, our goal is to choose the weights  $w_1, w_2, \dots, w_d$  and the bias  $w_0$  such that, on average, the predictions  $\hat{y}$  made by our model *best fit* the true labels  $y$  observed in the data
- models whose output prediction is determined by the affine transformation of input features are *linear models*, where the affine transformation is specified by the chosen weights and bias

## 1.2 Linear regression

- the linear regression model can be written more concisely in the vectorized form:

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \quad (2)$$

where

- $\mathbf{w}$  is the model's *weight vector*, containing the bias or intercept term  $w_0$  and the feature weights  $w_1, w_2, \dots, w_d$
- $\mathbf{x}$  is the instance's *feature vector*, containing  $x_0$  to  $x_d$ , where  $x_0$  is a "dummy feature" always equal to 1, which allows including the bias or intercept term into the weight vector
- in machine learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column
- $\mathbf{w}^\top$  is the *transpose* of  $\mathbf{w}$  (a row vector instead of a column vector) and  $\mathbf{w}^\top \mathbf{x}$  is the matrix multiplication of  $\mathbf{w}^\top$  and  $\mathbf{x}$ , which represents the scalar product of  $\mathbf{w}$  and  $\mathbf{x}$
- because the scalar product is symmetric, (2) can also be written as:

$$\hat{y} = \mathbf{x}^\top \mathbf{w}. \quad (3)$$

## 1.2 Linear regression

- in (3), the vector  $\mathbf{x}$  corresponds to features of a single data example
- we will often find it convenient to refer to features of our entire dataset of  $n$  examples via the *design matrix*  $\mathbf{X} \in \mathbb{R}^{n \times (d+1)}$
- here,  $\mathbf{X}$  contains one row for every example and one column for every feature:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \cdots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \vdots \\ \mathbf{x}^{(n)\top} \end{bmatrix}.$$

- for a collection of features  $\mathbf{X}$ , the predictions  $\hat{\mathbf{y}} \in \mathbb{R}^n$  can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}, \quad (4)$$

where  $\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \end{bmatrix}$

## 1.2 Linear regression

- given features of a training dataset  $\mathbf{X}$  and corresponding (known) labels  $\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$ , the goal of linear regression is to find the weight vector  $\mathbf{w}$  that given features of a new data example sampled from the same distribution as  $\mathbf{X}$ , the new example's label will (in expectation) be predicted with the lowest error
- even if we believe that the best model for predicting  $y$  given  $\mathbf{x}$  is linear, we would not expect to find a real-world dataset of  $n$  examples where  $y^{(i)}$  exactly equals  $\mathbf{w}^\top \mathbf{x}^{(i)}$ , for all  $1 \leq i \leq n$
- for example, whatever instruments we use to observe the features  $\mathbf{X}$  and labels  $\mathbf{y}$  might suffer small amount of measurement error
- thus, even when we are confident that the underlying relationship is linear, we will incorporate a *noise term* to account for such errors

## 1.2 Linear regression

- to refresh our memory, the probability density of a *normal distribution*  $\mathcal{N}(\mu, \sigma^2)$  with *mean*  $\mu$  and *variance*  $\sigma^2$  (standard deviation  $\sigma$ ) is given as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

- we assume that observations arise from noisy observations, where the noise is normally distributed as follows:

$$y = \mathbf{w}^\top \mathbf{x} + \epsilon,$$

where  $\epsilon \sim \mathcal{N}(0, \sigma^2)$

- thus, we can now write the *likelihood* of seeing a particular  $y^{(i)}$  for a given  $\mathbf{x}^{(i)}$  as:

$$p(y^{(i)} | \mathbf{x}^{(i)}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2\right).$$

## 1.2 Linear regression

- now, according to the principle of *maximum likelihood*, the best values of weights  $\mathbf{w}$  are those that maximize the likelihood of the entire dataset:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \prod_{i=1}^n p(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}). \quad (5)$$

- estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*
- while maximizing the product of many exponential functions might look difficult, we can simplify things significantly, without changing the objective, by maximizing the log of the likelihood instead
- for historical reasons, optimizations are more often expressed as minimization rather than maximization
- so, without changing anything, we can minimize the *negative log-likelihood* –  $-\log p(\mathbf{y}|\mathbf{X}, \mathbf{w})$ ; from (5), we have that:

$$-\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \sum_{i=1}^n \left[ \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2 \right].$$

## 1.2 Linear regression

- now we just need one more assumption that  $\sigma$  is some fixed constant; thus we can ignore the first term because it does not depend on  $\mathbf{w}$
- because the solution does not depend on  $\sigma$ , we can also ignore the multiplicative constant  $\frac{1}{\sigma^2}$
- as a consequence, negative log-likelihood is given as:

$$\sum_{i=1}^n \frac{1}{2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2.$$

- because  $\hat{y}^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)}$  are the predictions of the linear regression model, according to (4), the negative log-likelihood finally becomes:

$$\sum_{i=1}^n \frac{1}{2} (y^{(i)} - \hat{y}^{(i)})^2.$$

## 1.2 Linear regression

- on the other hand, before we start searching for the best weights (or model weights)  $\mathbf{w}$ , we will need two more things:
  - a quality measure for some given model
  - a procedure for updating the model to improve its quality
- in general, the quality measure for a certain model is given by the *loss function*, which quantifies the distance between the *real* and *predicted* value of the target
- the loss will usually be a non-negative number, where smaller values are better and perfect predictions have a loss of 0
- the most popular loss function in regression problems is the *squared error*
- when the model prediction for an example  $i$  is  $\hat{y}^{(i)}$  and the corresponding true label is  $y^{(i)}$ , the squared error is given by:

$$l^{(i)}(\mathbf{w}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2.$$

- the constant  $\frac{1}{2}$  makes no real difference, but will prove notationally convenient, canceling out when we take the derivative of the loss

## 1.2 Linear regression

- since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model weights
- to make things more concrete, consider the example below, where we plot a regression problem for a one-dimensional case, as shown in Figure 5

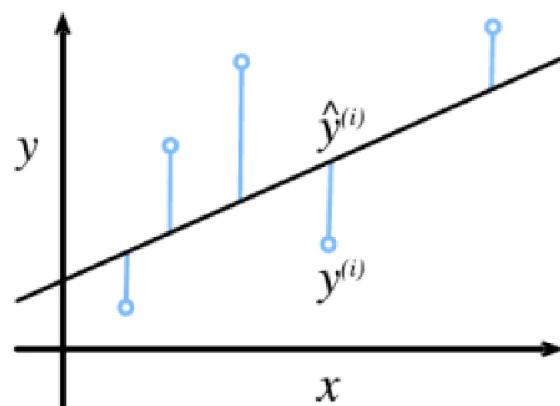


Figure 5: Linear regression with  $d = 1$ .

## 1.2 Linear regression

- note that large differences between estimates  $\hat{y}^{(i)}$  and observations  $y^{(i)}$  lead to even larger contributions to the loss, due to the quadratic dependence
- to measure the quality of a model on the entire dataset of  $n$  examples, we simply average (or equivalently, sum) the losses on the training set:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)})^2. \quad (6)$$

- this is called the *mean squared error loss function*; using (4), the mean squared error can be written as:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}).$$

- we can now see that, in the case of linear regression, the negative log-likelihood is *equal* (up to multiplicative constants) to the mean squared error loss function
- when training the model, we want to find parameters  $\mathbf{w}$  that minimize the total loss across all training examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathbf{w}).$$

## 1.2 Linear regression

- recall from first year calculus that, in order to find the minimum of the function  $\mathcal{L}(\mathbf{w})$ , we must compute its gradient, equal it with  $\mathbf{0}^\top$ , and extract the value of the minimum:

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left( \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \right) \\ &= \frac{1}{2n} \nabla_{\mathbf{w}} \left( \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{y}^\top \mathbf{y} \right) \\ &= \frac{1}{n} (\mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y}).\end{aligned}\tag{7}$$

- from  $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{0}^\top$ , we obtain:

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{0}^\top &\Leftrightarrow \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y} = \mathbf{0}^\top \\ &\Leftrightarrow \mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y} \\ &\Leftrightarrow \mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.\end{aligned}$$

- thus, for linear regression, we have the analytic (closed-form) solution:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

## 1.2 Linear regression

- this is called the *normal equation*
- the matrix  $\mathbf{X}^+ := (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$  is called the *pseudoinverse* or *Moore-Penrose inverse* of matrix  $\mathbf{X}$
- the pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix  $\mathbf{X}$  into the matrix multiplication of three matrices:  $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$
- the pseudoinverse is computed as  $\mathbf{X}^+ = \mathbf{V}\mathbf{S}^+\mathbf{U}^\top$ ; to compute the matrix  $\mathbf{S}^+$ , the algorithm takes  $\mathbf{S}$  and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix
- this approach is more efficient than computing the normal equation, plus it handles edge cases nicely: indeed, the normal equation may not work if the matrix  $\mathbf{X}^\top \mathbf{X}$  is not invertible (i.e., singular), such as if  $n < d$  or if some features are redundant, but the pseudoinverse is always defined

## 1.2 Linear regression

- the normal equation computes the inverse of  $\mathbf{X}^\top \mathbf{X}$ , which is an  $(d + 1) \times (d + 1)$  matrix
- the computational complexity of inverting such a matrix is typically about  $\mathcal{O}(d^{2.4})$  to  $\mathcal{O}(d^3)$ , depending on the implementation
- the SVD approach is about  $\mathcal{O}(d^2)$
- both the normal equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000)
- on the positive side, both are linear with regard to the number of instances in the training set (they are  $\mathcal{O}(n)$ ), so they handle large training sets efficiently, provided they can fit in memory

## 1.2 Linear regression

- also, once we have trained our linear regression model (using the normal equation or any other algorithm), predictions are very fast: the computational complexity is *linear* with regard to both the number of instances we want to make predictions on and the number of features
- in other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time
- next, we will look at a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 2

October 11th, 2022

## 1.3 Gradient descent

- *gradient descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- the general idea of gradient descent is to update the parameters iteratively in order to minimize the loss function
- suppose we are lost in the mountains in a dense fog, and we can only feel the slope of the ground below our feet; a good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the *steepest slope*
- this is exactly what gradient descent does: it measures the local gradient of the loss function with respect to the parameter vector  $w$ , and it goes in the direction of the steepest descent, which is given by the *negative gradient vector*, because the gradient points uphill
- once the gradient is zero, a minimum of the loss function has been reached

## 1.3 Gradient descent

- the steps of the algorithm are the following:

- initialize the values of the model weights, typically at random (this is called *random initialization*).
- iteratively update the parameters one small step at a time, in the direction of the negative gradient, until the algorithm *converges* to a minimum (see Figure 2). This is done by multiplying the gradient by a predetermined positive value  $\eta$  (called *learning rate*) and subtracting the resulting term from the current parameter values.

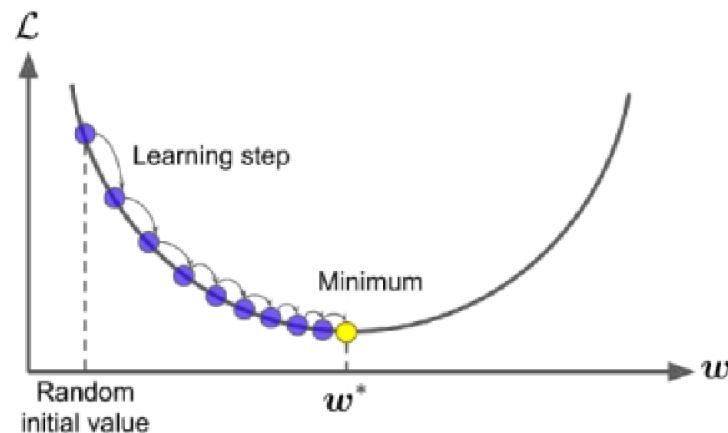


Figure 2: Gradient descent.

## 1.3 Gradient descent

- we can express the update mathematically as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}),$$

or, equivalently, as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left( \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}) \right),$$

where we used (6)

- according to (7), we can compute the gradient as follows:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} (\mathbf{X}^\top \mathbf{X} \mathbf{w} - \mathbf{X}^\top \mathbf{y}) = \frac{1}{n} \mathbf{X}^\top (\mathbf{X} \mathbf{w} - \mathbf{y}). \quad (8)$$

## 1.3 Gradient descent

- notice that this formula involves calculations over the full training set  $X$ , at each gradient descent step
- this is why the algorithm is called *batch gradient descent*: it uses the whole batch of training data at every step
- as a result it is very slow on very large training sets (but we will see much faster gradient descent algorithms shortly)
- however, gradient descent scales well with the number of features; training a linear regression model when there are hundreds of thousands of features is *much faster* using gradient descent than using the normal equation or SVD decomposition

## 1.3 Gradient descent

- an important parameter in gradient descent is the size of the steps, determined by the learning rate  $\eta$
- the value of the learning rate is manually pre-specified and not typically learned through model training
- this type of parameters that are tunable but not updated in the training loop are called *hyperparameters*
- *hyperparameter tuning* is the process by which hyperparameters are chosen, and typically requires that we adjust them based on the results of the training loop as assessed on a separate *validation dataset* (or *validation set*)

## 1.3 Gradient descent

- if the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 3)

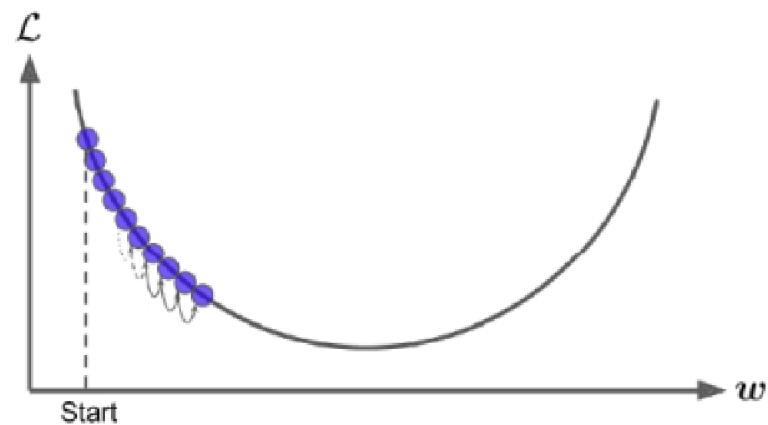


Figure 3: The learning rate is too small.

## 1.3 Gradient descent

- on the other hand, if the learning rate is too high, we might jump across the valley and end up on the other side, possibly even higher up than we were before
- this might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4)

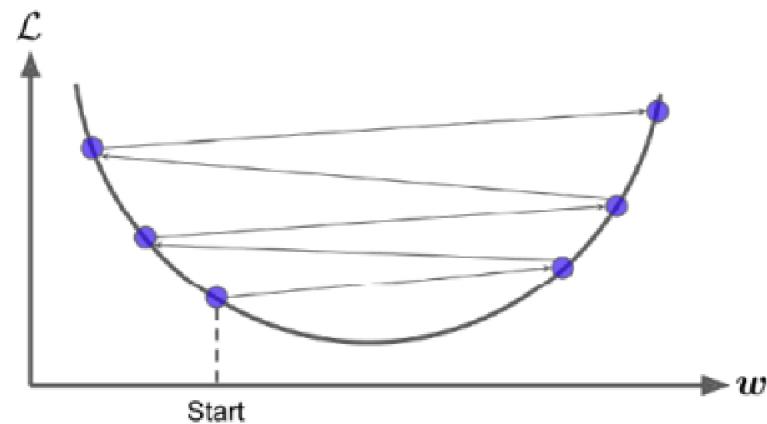


Figure 4: The learning rate is too big.

## 1.3 Gradient descent

- finally, not all loss functions look like nice, regular bowls
- there may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult
- Figure 5 shows the two main challenges with gradient descent
- if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*
- if it starts on the right, then it will take a very long time to cross the plateau
- and if we stop too early, we will never reach the global minimum

## 1.3 Gradient descent

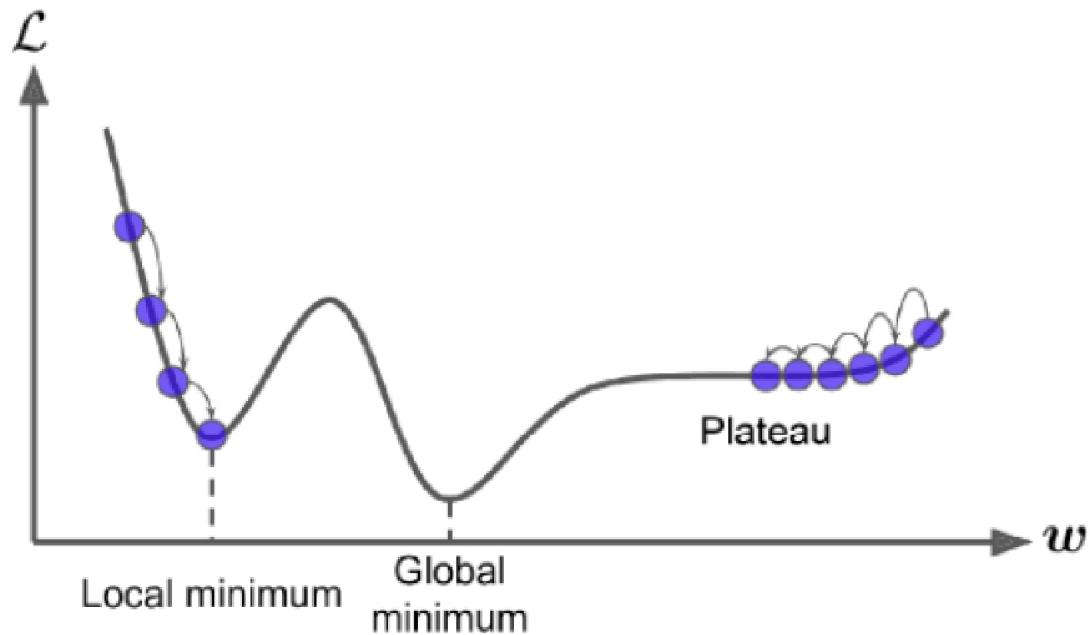


Figure 5: Gradient descent challenges.

## 1.3 Gradient descent

- fortunately, the mean squared error loss function for a linear regression model happens to be a *convex function*, which means that if we pick any two points on the curve, the line segment joining them never crosses the curve
- this implies that there are no local minima, just one global minimum
- it is also a continuous function with a slope that never changes abruptly
- these two facts have a great consequence: gradient descent is guaranteed to approach arbitrarily close the global minimum (if we wait long enough and if the learning rate is not too high)

## 1.3 Gradient descent

- in fact, the loss function has the shape of a bowl, but it can be an elongated bowl if the features have very *different scales*
- Figure 6 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right)

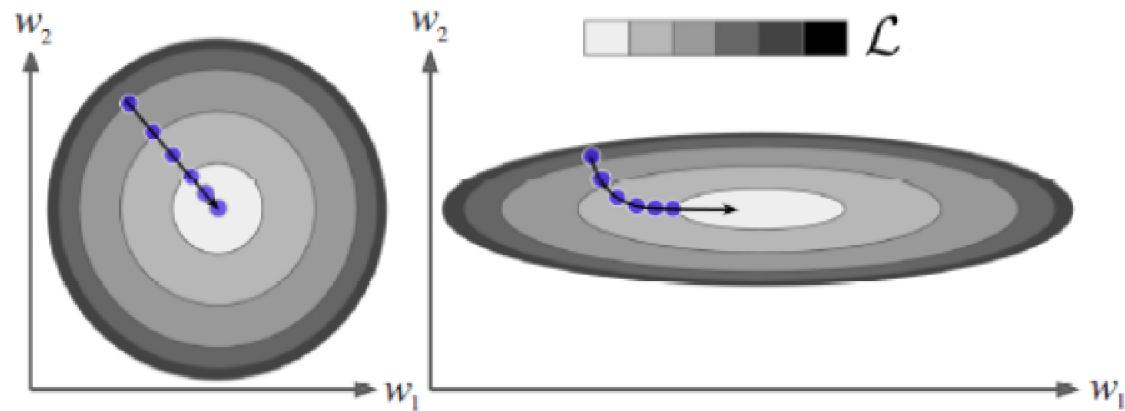


Figure 6: Gradient descent with (left) and without (right) feature scaling.

## 1.3 Gradient descent

- as we can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley
- it will eventually reach the minimum, but it will take a long time
- when using gradient descent, we should ensure that all features have a *similar scale*, or else it will take much longer to converge

## 1.3 Gradient descent

- the main problem with batch gradient descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large
- at the opposite extreme, *stochastic gradient descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance
- obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration
- it also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration
- mathematically, this can be expressed as:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} l^{(i)}(\mathbf{w}),$$

for some  $1 \leq i \leq n$

## 1.3 Gradient descent

- on the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the loss function will *bounce* up and down, decreasing only on average
- over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure 7)
- so once the algorithm stops, the final parameter values are good, but not optimal

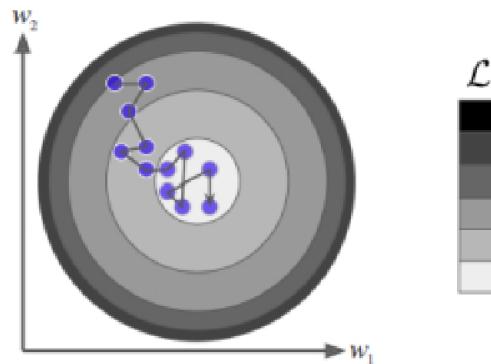


Figure 7: With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

## 1.3 Gradient descent

- when the loss function is very irregular (as in Figure 5), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does
- therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum
- one solution to this dilemma is to gradually reduce the learning rate
- the steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum
- the function that determines the learning rate at each iteration is called the *learning schedule*
- if the learning rate is reduced too quickly, we may get stuck in a local minimum, or even end up frozen halfway to the minimum
- if the learning rate is reduced too slowly, we may jump around the minimum for a long time and end up with a suboptimal solution if we halt training too early

## 1.3 Gradient descent

- by convention we iterate by rounds of  $n$  iterations; each round is called an *epoch*
- thus, in one epoch, each example will be used to update the weights of the model
- if we want to be sure that the algorithm goes through every instance at each epoch, we must shuffle the training set
- the last gradient descent algorithm we will look at is called *mini-batch gradient descent*
- it is simple to understand once we know batch and stochastic gradient descent: at each step, instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), mini-batch GD computes the gradients on small random sets of instances called *mini-batches*

## 1.3 Gradient descent

- the algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches
- as a result, mini-batch GD will end up walking around a bit closer to the minimum than stochastic GD – but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike linear regression)
- Figure 8 shows the paths taken by the three gradient descent algorithms in parameter space during training
- they all end up near the minimum, but batch GD's path actually stops at the minimum, while both stochastic GD and mini-batch GD continue to walk around
- however, we must not forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if we used a good learning schedule

## 1.3 Gradient descent

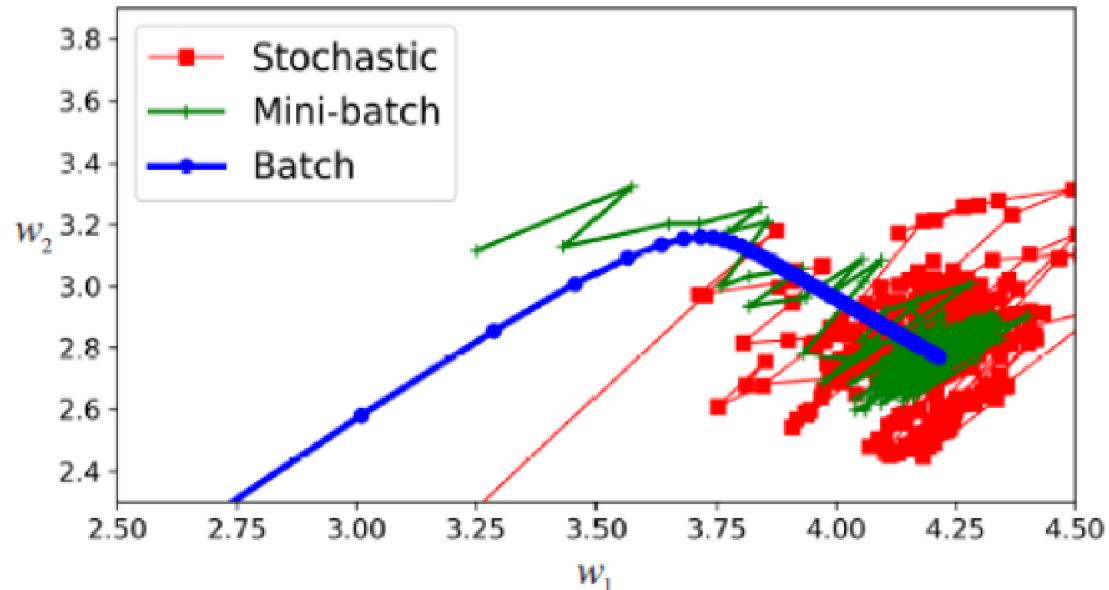


Figure 8: With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

## 1.3 Gradient descent

- in each iteration, we first randomly sample a mini-batch  $\mathcal{B}$  consisting of a fixed number of training examples
- we then compute the derivative (gradient) of the average loss on the mini-batch with regard to the model parameters
- finally, we multiply the gradient by a predetermined positive value and subtract the resulting term from the current parameter values
- we can express the update mathematically as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left( \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l^{(i)}(\mathbf{w}) \right).$$

- the set cardinality  $|\mathcal{B}|$  represents the number of examples in each mini-batch (the *batch size*); the batch size is a hyperparameter of the model

## 1.3 Gradient descent

- after training for some predetermined number of iterations (or until some other stopping criteria are met), we record the *estimated model weights*, denoted  $\hat{w}$
- note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards the minimizers, it cannot achieve it *exactly* in a finite number of steps
- linear regression happens to be a learning problem where there is only one minimum over the entire domain

## 1.3 Gradient descent

- however, for more complicated models, like deep networks, the loss surfaces contain many minima
- fortunately, for reasons that are not yet fully understood, deep learning practitioners rarely struggle to find parameters that minimize the loss *on training sets*
- the more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*
- given the learned linear regression model  $\hat{\mathbf{w}}^\top \mathbf{x}$ , we can now estimate target of a new sample (not contained in the training data) given its input features
- estimating targets given features is commonly called *prediction* or *inference*

## 1.4 Polynomial regression

- what if our data is more complex than a straight line?
- surprisingly, we can use a linear model to fit *nonlinear data*
- a simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features – this technique is called *polynomial regression*
- so far, we considered the linear regression setting described in (2), which allowed us to fit straight lines to data using maximum likelihood estimation
- however, straight lines are not sufficiently expressive when it comes to fitting more interesting data
- fortunately, linear regression offers us a way to fit nonlinear functions within the linear regression framework: since “linear regression” only refers to “linear in the parameters”, we can perform an arbitrary nonlinear transformation  $\phi(\mathbf{x})$  of the inputs  $\mathbf{x}$  and then linearly combine the components of this transformation

## 1.4 Polynomial regression

- the corresponding linear regression model is:

$$\hat{y} = \mathbf{w}^\top \phi(\mathbf{x}), \quad (9)$$

where  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$  is a (nonlinear) transformation of the inputs  $\mathbf{x}$  and  $\phi_k : \mathbb{R}^d \rightarrow \mathbb{R}$  is the  $k$ th component of the feature vector  $\phi$

- note that the feature vector model parameters still appear only linearly
- note that when there are multiple features, polynomial regression is capable of finding relationships between features (which is something a plain linear regression model cannot do)
- this is made possible by the fact that we can also add all combinations of features up to the given degree
- for example, if there were two features  $a$  and  $b$ , we would not only add the features  $a^2$ ,  $a^3$ ,  $b^2$ , and  $b^3$ , but also the combinations  $ab$ ,  $a^2b$ , and  $ab^2$

## 1.4 Polynomial regression

- consider the dataset in Figure 9; the dataset consists of  $n = 10$  pairs  $(x^{(i)}, y^{(i)})$ , where  $x^{(i)} \sim \mathcal{U}[-5, 5]$  and

$$y^{(i)} = \sin\left(\frac{x^{(i)}}{5}\right) + \cos(x^{(i)}) + \epsilon,$$

where  $\epsilon \sim \mathcal{N}(0, 0.2^2)$

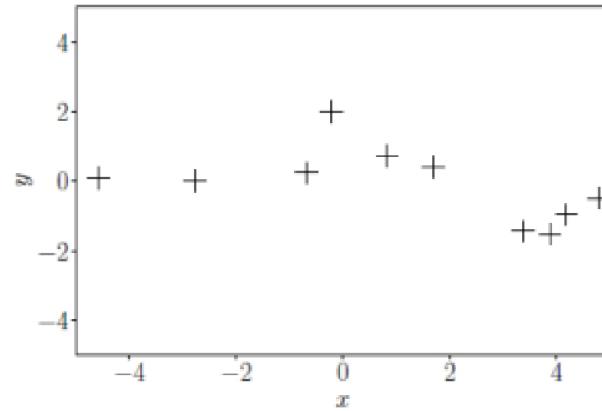


Figure 9: Polynomial regression: dataset consisting of  $(x^{(i)}, y^{(i)})$  pairs,  $i = 1, \dots, 10$ .

## 1.4 Polynomial regression

- assume that we use a polynomial model

$$\hat{y} = \mathbf{w}^\top \phi(x),$$

to fit it, where  $\mathbf{w} \in \mathbb{R}^M$ ,

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^{M-1} \end{bmatrix} \in \mathbb{R}^M.$$

- we can evaluate the quality of a model by computing the loss  $\mathcal{L}$
- instead of using this squared loss, we often use the *root mean square error (RMSE)*:

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \phi(\mathbf{x}^{(i)}) - y^{(i)})^2},$$

which

- allows us to compare errors of datasets with different sizes
- has the same scale and the same units as the observed function values  $y^{(i)}$

## 1.4 Polynomial regression

- in order to select the best model to fit our dataset, we can use the RMSE to determine the best degree of the polynomial by finding the polynomial degree  $M$  that minimizes the RMSE
- given that the polynomial degree is a natural number, we can perform a brute-force search and enumerate all (reasonable) values of  $M$
- for a training set of size  $n$  it is sufficient to test  $0 \leq M \leq n - 1$
- for  $M < n$ , the maximum likelihood estimator is unique
- for  $M \geq n$ , we have more parameters than data points, and would need to solve an underdetermined system of linear equations, so that there are infinitely many possible maximum likelihood estimators

## 1.4 Polynomial regression

- Figure 10 shows a number of polynomial fits determined by polynomial regression for the dataset from Figure 9 with  $n = 10$  observations
- we notice that polynomials of low degree (e.g., constants ( $M = 0$ ) or linear ( $M = 1$ )) fit the data poorly and, hence, are poor representations of the true underlying function
- for degrees  $M = 3, \dots, 6$ , the fits look plausible and smoothly interpolate the data
- when we go to higher-degree polynomials, we notice that they fit the data better and better
- in the extreme case of  $M = n - 1 = 9$ , the function will pass through every single data point
- however, these high-degree polynomials oscillate wildly and are a poor representation of the underlying function that generated the data, such that we suffer from *overfitting*

## 1.4 Polynomial regression

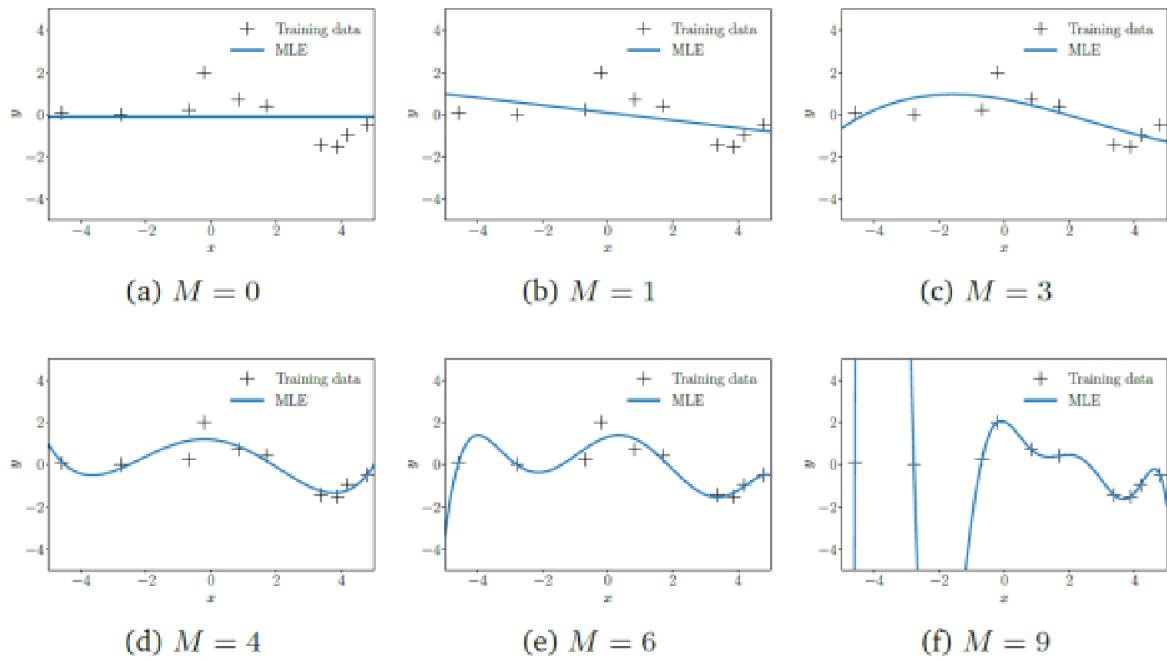


Figure 10: Maximum likelihood fits for different polynomial degrees  $M$ .

## 1.4 Polynomial regression

- however, the goal is to achieve good *generalization* by making accurate predictions for new (unseen) data
- we obtain some quantitative insight into the dependence of the generalization performance on the polynomial of degree  $M$  by considering a separate test set comprising 200 data points generated using exactly the same procedure used to generate the training set
- as test inputs, we choose a linear grid of 200 points in the interval of  $[-5, 5]$
- for each choice of  $M$ , we evaluate the RMSE for both the training data and the test data

## 1.4 Polynomial regression

- looking now at the *test error*, which is a qualitative measure of the generalization properties of the corresponding polynomial, we notice that initially the test error decreases; see Figure 11 (orange)
- for fourth-order polynomials, the test error is relatively low and stays relatively constant up to degree 5
- however, from degree 6 onward, the test error increases significantly, and high-order polynomials have very bad generalization properties
- in this particular example, this also is evident from the corresponding maximum likelihood fits in Figure 10

## 1.4 Polynomial regression

- note that the *training error* (blue curve in Figure 11) never increases when the degree of the polynomial increases
- in our example, the best generalization (the point of the smallest *test error*) is obtained for a polynomial of degree  $M = 4$

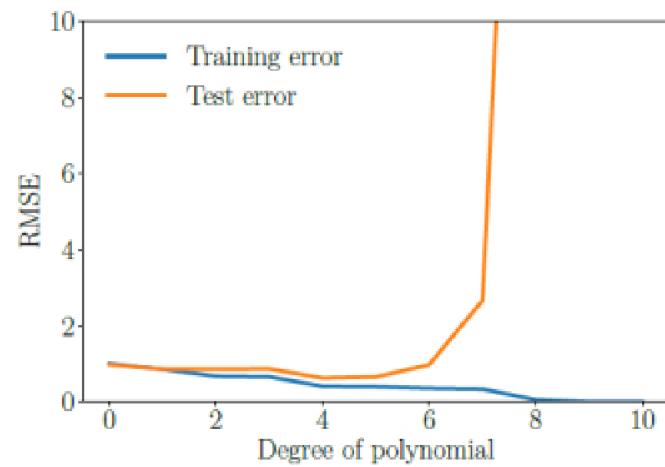


Figure 11: Training and test error.

## 1.4 Polynomial regression

- an important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:
  - ① *Bias*. This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.
  - ② *Variance*. This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.
  - ③ *Irreducible error*. This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).
- increasing a model's complexity will typically increase its variance and reduce its bias
- conversely, reducing a model's complexity increases its bias and reduces its variance
- this is why it is called the *bias/variance trade-off*

## 1.5 Regularized linear models

- we just saw that maximum likelihood estimation is prone to overfitting
- we often observe that the magnitude of the parameter values becomes *relatively large* if we run into overfitting
- to mitigate the effect of huge parameter values, we can place a *prior distribution*  $p(\mathbf{w})$  on the parameters
- the prior distribution explicitly encodes what parameter values are plausible (before having seen any data)
- for example, a Gaussian prior  $\mathbf{w} \sim \mathcal{N}(0, 1)$  on a single parameter encodes that parameter values are expected to lie in the interval  $[-2, 2]$  (two standard deviations around the mean value)

## 1.5 Regularized linear models

- once a dataset  $\mathbf{X}, \mathbf{y}$  is available, instead of maximizing the likelihood, we seek parameters that maximize the posterior distribution  $p(\mathbf{w}|\mathbf{X}, \mathbf{y})$
- this procedure is called *maximum a posteriori* estimation
- the posterior over the weights  $\mathbf{w}$ , given the training data  $\mathbf{X}, \mathbf{y}$ , is obtained by applying Bayes' theorem as:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{w})p(\mathbf{w})}{p(\mathbf{y}|\mathbf{X})}.$$

- to find the maximum a posteriori estimation, we minimize the *negative log-posterior distribution* with respect to  $\mathbf{w}$ :

$$-\log p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = -\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) - \log p(\mathbf{w}) + \log p(\mathbf{y}|\mathbf{X}). \quad (10)$$

- we see that the log-posterior in (10) is the sum of the log-likelihood  $\log p(\mathbf{y}|\mathbf{X}, \mathbf{w})$  and the log-prior  $\log p(\mathbf{w})$ , so that the maximum a posteriori estimate will be a “compromise” between the prior (our suggestion for plausible parameter values before observing data) and the data-dependent likelihood

## 1.5 Regularized linear models

- first, assume that the weights are normally distributed with variance  $\tau$ ,  $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \tau^2 \mathbf{I})$ , which implies that:

$$-\log p(\mathbf{w} | \mathbf{X}, \mathbf{y}) = \frac{1}{2\sigma^2} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2\tau^2} \mathbf{w}^\top \mathbf{w} + \text{const.} \quad (11)$$

where const. represent terms that are independent of  $\mathbf{w}$

- here, the first term corresponds to the contribution from the log-likelihood, and the second term originates from the log-prior
- by taking the gradient of (11) and making it equal with  $\mathbf{0}^\top$ , the maximum a posteriori estimate for  $\mathbf{w}$  will be:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y},$$

where we denoted  $\lambda := \frac{\sigma^2}{\tau^2}$

## 1.5 Regularized linear models

- similarly as for maximum likelihood estimation, we can define a loss corresponding to maximum a posteriori estimation:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2} \lambda \|\mathbf{w}\|_2^2,$$

where  $\|\mathbf{w}\|_2 = \left( \sum_{j=0}^d |w_j|^2 \right)^{1/2}$  is the  $\ell_2$  norm of  $\mathbf{w}$

- linear regression using this loss function is called *ridge regression*
- ridge regression is a *regularized* version of linear regression: a regularization term equal to  $\frac{1}{2} \lambda \|\mathbf{w}\|_2^2$  is added to the loss function
- this forces the learning algorithm to not only fit the data but also keep the model weights as *small* as possible
- note that the regularization term should only be added to the loss function during training
- once the model is trained, we want to use the unregularized performance measure to evaluate the model's performance

## 1.5 Regularized linear models

- the hyperparameter  $\lambda$  controls how much we want to regularize the model
- if  $\lambda = 0$ , then ridge regression is just linear regression
- if  $\lambda$  is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean
- as with linear regression, we can perform ridge regression either by computing a closed-form equation or by performing gradient descent
- the pros and cons are the same; for gradient descent, we just add  $\lambda\mathbf{w}$  to the gradient vector:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^T (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}.$$

- it is important to *scale* the data before performing ridge regression, as it is sensitive to the scale of the input features, which is true of most regularized models

## 1.5 Regularized linear models

- on the other hand, we can assume a Laplace distribution for the weights,  $\mathbf{w} \sim \text{Lap}(\mathbf{0}, bI)$
- the probability density for the Laplace distribution is given as:

$$p(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{1}{b}|x - \mu|\right).$$

- here,  $\mu$  is a location parameter and  $b > 0$  is a scale parameter
- the loss corresponding to the Laplace prior on the weights is:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda \|\mathbf{w}\|_1,$$

where  $\|\mathbf{w}\|_1 = \sum_{j=0}^d |w_j|$  is the  $\ell_1$  norm of  $\mathbf{w}$

- linear regression using this loss function is called *lasso regression*

## 1.5 Regularized linear models

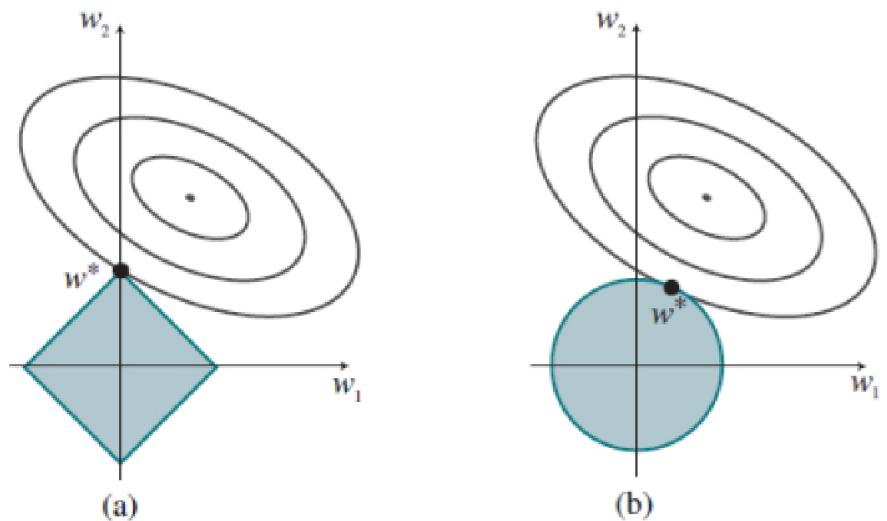
- *Least Absolute Shrinkage and Selection Operator Regression* (usually simply called lasso regression) is another regularized version of linear regression: just like ridge regression, it adds a regularization term to the loss function, but it uses the  $\ell_1$  norm of the weight vector instead of half the square of the  $\ell_2$  norm
- an important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero)
- in other words, lasso regression automatically performs *feature selection* and outputs a *sparse model*
- that is, it often sets many weights to zero, effectively declaring the corresponding attributes to be completely irrelevant
- models that discard attributes can be easier for a human to understand, and may be less likely to overfit

## 1.5 Regularized linear models

- Figure 12 gives an intuitive explanation of why  $\ell_1$  regularization leads to weights of zero, while  $\ell_2$  regularization does not
- note that minimizing  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\lambda\|\mathbf{w}\|_2^2$  or  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\|\mathbf{w}\|_1$  is equivalent to minimizing  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$  subject to the constraint that  $\frac{1}{2}\lambda\|\mathbf{w}\|_2^2 \leq c$  or  $\lambda\|\mathbf{w}\|_1 \leq c$ , for some constant  $c$  that is related to  $\lambda$
- now, in Figure 12(a) the diamond-shaped box represents the set of points  $\mathbf{w}$  in two-dimensional weight space that have  $\ell_1$  norm less than  $c$ ; our solution will have to be somewhere inside this box
- the concentric ovals represent contours of the mean squared error function, with the minimum loss at the center

- we want to find the point in the box that is *closest* to the minimum; we can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy
- and of course the corners are the points that have a value of zero in some dimension
- in Figure 12(b), we have done the same for the  $\ell_2$  norm, which represents a circle rather than a diamond
- here we can see that, in general, there is no reason for the intersection to appear on one of the axes; thus  $\ell_2$  regularization does not tend to produce zero weights

## 1.5 Regularized linear models



**Figure 12:** Why  $\ell_1$  regularization tends to produce a sparse model. Left: with  $\ell_1$  regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: with  $\ell_2$  regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

## 1.5 Regularized linear models

- the lasso loss function is not differentiable whenever  $w_j = 0$  for some  $j = 0, 1, \dots, d$ , but gradient descent still works fine if we use a *subgradient vector*  $\mathbf{g}$  instead, when any  $w_j = 0$
- we can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point
- a subgradient vector we can use for gradient descent with the lasso loss function is:

$$\mathbf{g}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda \text{sign}(\mathbf{w}),$$

where  $\text{sign}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \\ +1, & \text{if } x > 0 \end{cases}$

## 1.5 Regularized linear models

- *elastic net* is a middle ground between ridge regression and lasso regression
- the regularization term is a simple mix of both ridge and lasso's regularization terms, and we can control the mix ratio  $r \in [0, 1]$
- when  $r = 0$ , elastic net is equivalent to ridge regression, and when  $r = 1$ , it is equivalent to lasso regression:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + r\lambda \|\mathbf{w}\|_1 + \frac{1-r}{2}\lambda \|\mathbf{w}\|_2^2.$$

## 1.5 Regularized linear models

- so when should we use plain linear regression (i.e., without any regularization), ridge, lasso, or elastic net?
- it is almost always preferable to have at least a *little bit of regularization*, so generally we should avoid plain linear regression
- ridge is a good default, but if we suspect that only a few features are useful, we should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as we have discussed
- in general, elastic net is preferred over lasso, because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated

- a very different way to *regularize* iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum; this is called *early stopping*
- Figure 13 shows a complex model (in this case, a high-degree polynomial regression model) being trained with batch gradient descent
- as the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set
- after a while though, the validation error stops decreasing and starts to go back up
- this indicates that the model has started to overfit the training data

## 1.5 Regularized linear models

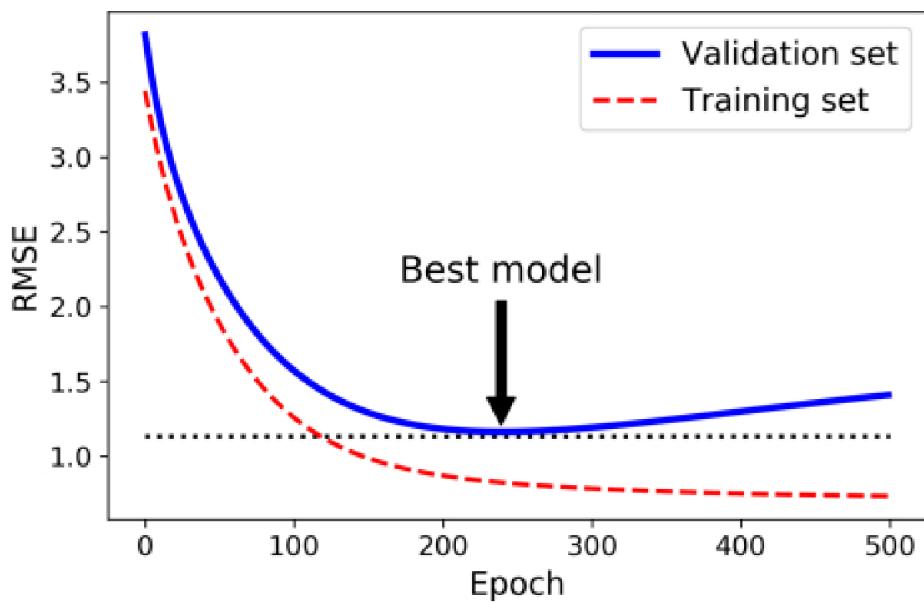


Figure 13: Early stopping regularization.

- with early stopping, we just stop training as soon as the validation error reaches the minimum
- with stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether we have reached the minimum or not
- one solution is to stop only after the validation error has been above the minimum for some time (when we are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum

## 1.6 Logistic regression

- some regression algorithms can be used for classification (and vice versa)
- *logistic regression* (also called *logit regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?)
- if the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”)
- this makes it a *binary classifier*

## 1.6 Logistic regression

- just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly, like the linear regression model does, it outputs the *logistic* of this result:

$$\hat{p} = \sigma(\mathbf{w}^T \mathbf{x}).$$

- the logistic, denoted  $\sigma(\cdot)$ , is the *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1; it is defined as:

$$\sigma(t) = \frac{1}{1 + \exp(-t)},$$

and its graph is given in Figure 14

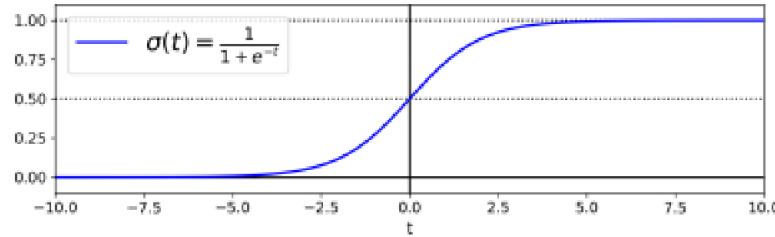


Figure 14: Sigmoid function.

## 1.6 Logistic regression

- once the logistic regression model has estimated the probability  $\hat{p}$  that an instance  $\mathbf{x}$  belongs to the positive class, it can make its prediction  $\hat{y}$  easily as:

$$\hat{y} = \begin{cases} 0, & \text{if } \hat{p} < 0.5 \\ 1, & \text{if } \hat{p} \geq 0.5 \end{cases}.$$

- notice that  $\sigma(t) < 0.5$  when  $t < 0$ , and  $\sigma(t) \geq 0.5$  when  $t \geq 0$ , so a logistic regression model predicts 1 if  $\mathbf{w}^T \mathbf{x}$  is positive and 0 if it is negative
- the score  $t$  is often called the *logit*; the name comes from the fact that the *logit function*, defined as  $\text{logit}(\hat{p}) = \log(\hat{p}/(1 - \hat{p}))$ , is the inverse of the sigmoid function
- indeed, if we compute the logit of the estimated probability  $\hat{p}$ , we will find that the result is  $t$
- the logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class

## 1.6 Logistic regression

- for training the logistic regression, the objective is to set the weight vector  $w$  so that the model estimates high probabilities for positive instances ( $y = 1$ ) and low probabilities for negative instances ( $y = 0$ )
- this idea is captured by the loss function shown below, for a single training instance  $x$ :

$$l(w) = \begin{cases} -\log \hat{p}, & \text{if } y = 1 \\ -\log(1 - \hat{p}), & \text{if } y = 0 \end{cases}$$

- this loss function makes sense because  $-\log t$  grows very large when  $t$  approaches 0, so the loss will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance
- on the other hand,  $-\log t$  is close to 0 when  $t$  is close to 1, so the loss will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want

## 1.6 Logistic regression

- it can be also written as:

$$l(\mathbf{w}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p}).$$

- the loss function over the whole training set is the average loss over all training instances
- it can be written in a single expression called the *log loss*, as:

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^n \left[ y^{(i)} \log \hat{p}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]. \quad (12)$$

- the bad news is that there is no known closed-form equation to compute the value of  $\mathbf{w}$  that minimizes this loss function (there is no equivalent of the normal equation)
- the good news is that this loss function is convex, so gradient descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and we wait long enough)

## 1.6 Logistic regression

- the gradient of the loss function is given by:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^\top (\sigma(\mathbf{X}\mathbf{w}) - \mathbf{y}).$$

- this equation looks very much like equation (8): for each instance it computes the prediction error and multiplies it by the feature values, and then it computes the average over all training instances
- once we have the gradient vector containing all the partial derivatives, we can use it in the batch gradient descent algorithm
- for stochastic GD, we would take one instance at a time, and for mini-batch GD we would use a mini-batch at a time
- just like the other linear models, logistic regression models can be regularized using  $\ell_1$  or  $\ell_2$  penalties

## 1.7 Softmax regression

- the logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers
- this is called *softmax regression*, or *multinomial logistic regression*
- assume we have a classification problem with  $q$  classes
- the idea is simple: when given an instance  $\mathbf{x}$ , the softmax regression model first computes a score  $o_k(\mathbf{x})$  for each class  $k = 1, 2, \dots, q$ , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores
- the equation to compute  $o_k(\mathbf{x})$  should look familiar, as it is just like the equation for linear regression prediction:

$$o_k(\mathbf{x}) = \mathbf{w}_k^\top \mathbf{x}.$$

## 1.7 Softmax regression

- note that each class has its own dedicated parameter vector  $\mathbf{w}_k \in \mathbb{R}^d$
- all these vectors are typically stored as columns in a parameter matrix  $\mathbf{W} \in \mathbb{R}^{d \times q}$ :

$$\mathbf{W} = [ \mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_q ].$$

- once we have computed the score of every class for the instance  $\mathbf{x}$ , we can estimate the probability  $\hat{p}_k$  that the instance belongs to class  $k$  by running the scores through the softmax function:

$$\hat{p}_k = \text{softmax}(\mathbf{o}(\mathbf{x}))_k = \frac{\exp(o_k(\mathbf{x}))}{\sum_{l=1}^q \exp(o_l(\mathbf{x}))},$$

where

- $\mathbf{o}(\mathbf{x})$  is a vector containing the scores of each class for the instance  $\mathbf{x}$
- $\text{softmax}(\mathbf{o}(\mathbf{x}))_k$  is the estimated probability that the instance  $\mathbf{x}$  belongs to class  $k$ , given the scores of each class for that instance

## 1.7 Softmax regression

- the function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials)
- the scores are generally called *logits* or *log-odds* (although they are actually *unnormalized log-odds*)
- just like the logistic regression classifier, the softmax regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score):

$$\hat{y} = \operatorname{argmax}_k \text{softmax}(\mathbf{o}(\mathbf{x}))_k = \operatorname{argmax}_k o_k(\mathbf{x}) = \operatorname{argmax}_k \mathbf{w}_k^\top \mathbf{x}.$$

- the argmax operator returns the value of a variable that maximizes a function
- in this equation, it returns the value of  $k$  that maximizes the estimated probability  $\text{softmax}(\mathbf{o}(\mathbf{x}))_k$
- the softmax regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants, not multiple people in one picture

## 1.7 Softmax regression

- for training, the objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes)
- minimizing the loss function shown below, called the *cross entropy*, should lead to this objective, because it penalizes the model when it estimates a low probability for a target class
- cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^q y_k^{(i)} \log \hat{p}_k^{(i)},$$

where

- $y_k^{(i)}$  is the target probability that the  $i$ th instance belongs to class  $k$ . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.
- notice that when there are just two classes ( $q = 2$ ), this loss function is equivalent to the logistic regression's loss function in (12)

## 1.7 Softmax regression

- the cross entropy between probability distributions  $p$  and  $q$ , denoted  $H(p, q)$ , measures the *mismatch* between probabilities  $q$  upon seeing data that were actually generated according to probabilities  $p$ :

$$H(p, q) = - \sum_x p(x) \log q(x).$$

- the lowest possible cross entropy is achieved when  $p = q$ ; in this case, the cross-entropy between  $p$  and  $q$  is:

$$H(p, p) = H(p) = - \sum_x p(x) \log p(x),$$

i.e., cross entropy is equal to the entropy of probability distribution  $p$

- the gradient of the cross entropy loss function with regard to  $\mathbf{w}_k$  is given by:

$$\nabla_{\mathbf{w}_k} \mathcal{L}(\mathbf{W}) = \frac{1}{n} \mathbf{X}^\top (\hat{\mathbf{p}}_k - \mathbf{y}_k) = \frac{1}{n} \mathbf{X}^\top (\text{softmax}(\mathbf{o}(\mathbf{x}))_k - \mathbf{y}_k).$$

- now we can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix  $\mathbf{W}$  that minimizes the loss function

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 3

October 18th, 2022

## 2 Multilayer perceptrons

### 2.1 From linear models to deep neural networks

- in Chapter 1, we only talked about linear models
- while *neural networks* cover a much richer family of models, we can begin thinking of the linear model as a neural network by expressing it in the language of neural networks
- to begin, let us start by rewriting things in a “layer” notation
- diagrams are used in deep learning to visualize models
- in Figure 1, we depict the linear regression model as a neural network
- note that these diagrams highlight the connectivity pattern, such as how each input is connected to the output, but not the values taken by the weights or biases

## 2.1 From linear models to deep neural networks

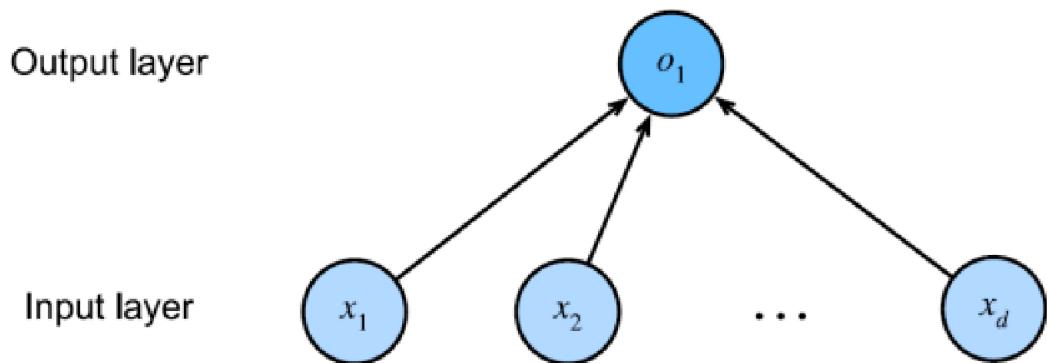


Figure 1: Linear regression is a single-layer neural network.

## 2.1 From linear models to deep neural networks

- for the neural network shown in Figure 1, the inputs are  $x_1, \dots, x_d$ , so the *number of inputs* (or *feature dimensionality*) in the input layer is  $d$
- the output of the network in Figure 1 is  $o_1$ , so the *number of outputs* in the output layer is 1
- note that the input values are all *given* and there is just a single *computed* neuron
- focusing on where computation takes place, conventionally, we do not consider the input layer when counting layers
- that is to say, the *number of layers* for the neural network in Figure 1 is 1
- we can think of linear regression models as neural networks consisting of just a single artificial neuron, or as *single-layer neural networks*

## 2.1 From linear models to deep neural networks

- since, for linear regression, every input is connected to every output (in this case, there is only one output), we can regard this transformation (the output layer in Figure 1) as a *fully-connected layer* or *dense layer*
- we will talk a lot more about networks composed of such layers in this and the following chapters
- since linear regression (invented in 1795) predates computational neuroscience, linear regression was not originally described as a neural network
- to see why linear models were a natural place to begin when the cyberneticists/neurophysiologists Warren McCulloch and Walter Pitts began to develop models of artificial neurons, consider the cartoonish picture of a biological neuron in Figure 2, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*

## 2.1 From linear models to deep neural networks

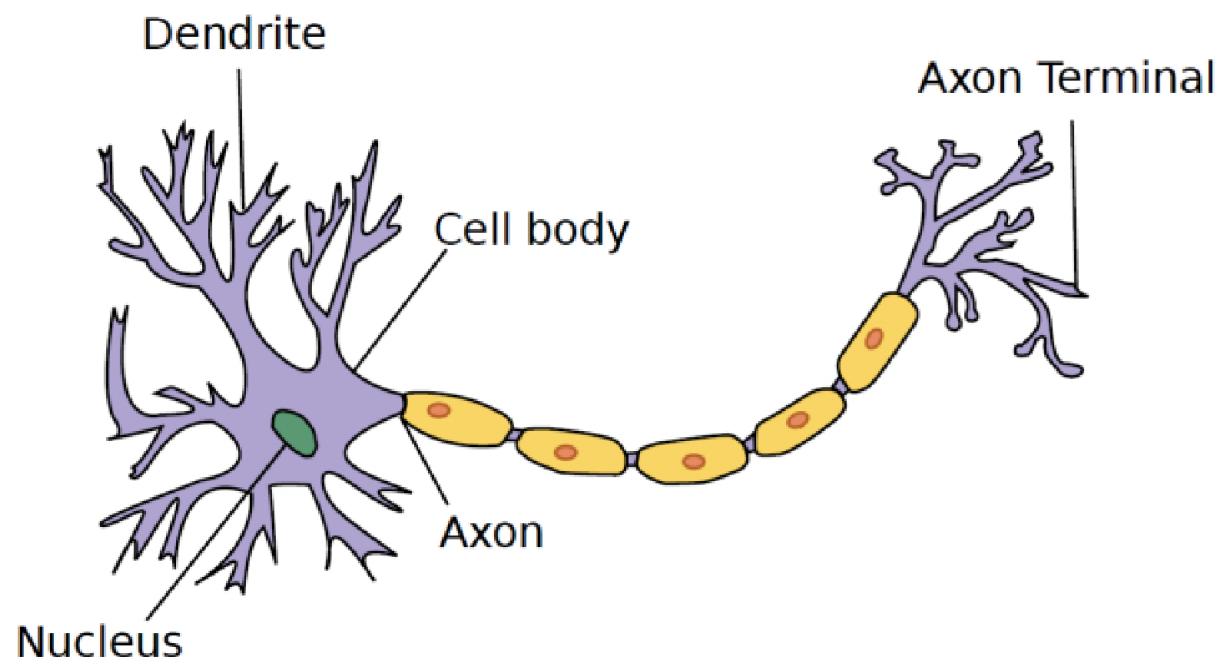


Figure 2: The real neuron.

## 2.1 From linear models to deep neural networks

- information  $x_i$  arriving from other neurons (or environmental sensors, such as the retina) is received in the dendrites
- in particular, that information is weighted by *synaptic weights*  $w_i$ , determining the effect of the inputs (e.g., activation or inhibition via the product  $x_i w_i$ )
- the weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum  $y = \sum_{i=1}^d x_i w_i + b$ , and this information is then sent for further processing in the axon  $y$ , typically after some nonlinear processing via  $\sigma(y)$
- from there, it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites

## 2.1 From linear models to deep neural networks

- certainly, the high-level idea that many such units could be stitched together with the right connectivity and right learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express, owes to our study of real biological neural systems
- at the same time, most research in deep learning today draws little direct inspiration from neuroscience
- although airplanes might have been *inspired* by birds, ornithology has not been the primary driver of aeronautics innovation for many years
- likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, statistics, and computer science

## 2.1 From linear models to deep neural networks

- softmax regression can also be framed as a neural network
- let us start off with a simple image classification problem; here, each input consists of a  $2 \times 2$  grayscale image
- we can represent each pixel value with a single scalar, giving us four features  $x_1, x_2, x_3, x_4$
- further, let us assume that each image belongs to one among the categories "cat", "chicken", and "dog"
- next, we have to choose how to represent the labels; we have two obvious choices
- perhaps the most natural impulse would be to choose  $y \in \{1, 2, 3\}$ , where the integers represent {dog, cat, chicken}, respectively; this is a great way of *storing* such information on a computer
- if the categories had some natural ordering among them, say if we were trying to predict {baby, toddler, adolescent, young adult, adult, geriatric}, then it might even make sense to cast this problem as regression, and keep the labels in this format

## 2.1 From linear models to deep neural networks

- but general classification problems do not come with natural orderings among the classes
- fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*
- a one-hot encoding is a vector with as many components as we have categories
- the component corresponding to a particular instance's category is set to 1, and all other components are set to 0
- in our case, a label  $y$  would be a three-dimensional vector, with  $(1, 0, 0)$  corresponding to "cat",  $(0, 1, 0)$  to "chicken", and  $(0, 0, 1)$  to "dog":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

## 2.1 From linear models to deep neural networks

- in order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class
- to address classification with linear models, we will need as many affine functions as we have outputs
- each output will correspond to its own affine function
- in our case, since we have 4 features and 3 possible output categories, we will need 12 scalars to represent the weights ( $w$  with subscripts), and 3 scalars to represent the biases ( $b$  with subscripts)
- we compute these three *logits*,  $o_1$ ,  $o_2$ , and  $o_3$ , for each input:

$$\begin{aligned}o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.\end{aligned}$$

## 2.1 From linear models to deep neural networks

- we can depict this calculation with the neural network diagram shown in Figure 3
- just as in linear regression, softmax regression is also a *single-layer neural network*
- and, since the calculation of each output,  $o_1$ ,  $o_2$ , and  $o_3$ , depends on all inputs,  $x_1$ ,  $x_2$ ,  $x_3$ , and  $x_4$ , the output layer of softmax regression can also be described as a fully-connected layer

## 2.1 From linear models to deep neural networks

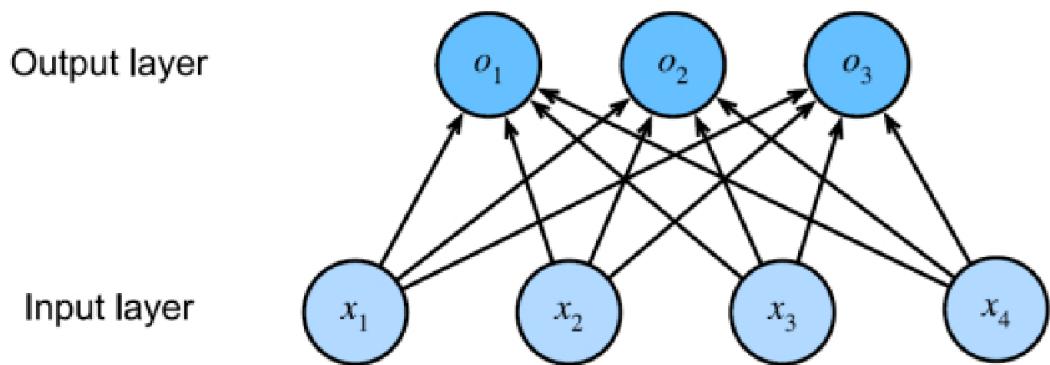


Figure 3: Softmax regression is a single-layer neural network.

## 2.1 From linear models to deep neural networks

- to express the model more compactly, we use the linear algebra notation
- in vector form, we arrive at  $\mathbf{o} = \mathbf{Wx} + \mathbf{b}$ , a form better suited both for mathematics, and for writing code
- note that we have gathered all of our weights into a  $3 \times 4$  matrix, and that, for features of a given data example  $\mathbf{x}$ , our outputs are given by a matrix-vector product of our weights by our input features, plus our biases  $\mathbf{b}$
- as we will see in this and subsequent chapters, fully-connected layers are ubiquitous in deep learning
- however, as the name suggests, fully-connected layers are *fully* connected, with potentially many learnable parameters
- specifically, for any fully-connected layer with  $d$  inputs and  $q$  outputs, the parameterization cost is  $\mathcal{O}(dq)$ , which can be prohibitively high in practice

## 2.1 From linear models to deep neural networks

- as we discussed in Chapter 1, we want to interpret the outputs of our model as probabilities
- we do this by applying the *softmax function* to the logits vector  $\mathbf{o}$ :

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_{j'=1}^q \exp(o_{j'})}.$$

- it is easy to see that  $\sum_{j=1}^q \hat{y}_j = 1$ , with  $0 \leq \hat{y}_j \leq 1$ , for all  $1 \leq j \leq q$
- thus,  $\hat{\mathbf{y}}$  is a proper *probability distribution*, whose element values can be interpreted accordingly
- note that the softmax operation does not change the ordering among the logits  $\mathbf{o}$ , which are simply the pre-softmax values that determine the probabilities assigned to each class
- therefore, during prediction, we can still pick out the most likely class by:

$$\underset{j}{\operatorname{argmax}} \hat{y}_j = \underset{j}{\operatorname{argmax}} o_j.$$

## 2.1 From linear models to deep neural networks

- although softmax is a nonlinear function, the outputs of softmax regression are still *determined* by an affine transformation of input features; thus, softmax regression is a linear model
- assume that we are given a design matrix  $\mathbf{X}$  of  $n$  examples with feature dimensionality (number of inputs)  $d$
- assume that we have  $q$  categories in the output
- then, the design matrix  $\mathbf{X} \in \mathbb{R}^{n \times d}$ , weights  $\mathbf{W} \in \mathbb{R}^{d \times q}$ , and the bias satisfies  $\mathbf{b} \in \mathbb{R}^{1 \times q}$
- thus, we have:

$$\begin{aligned}\mathbf{O} &= \mathbf{XW} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}).\end{aligned}$$

## 2.1 From linear models to deep neural networks

- this accelerates the dominant operation into a matrix-matrix product  $\mathbf{X}\mathbf{W}$  vs. the matrix-vector products we would be executing if we processed one example at a time
- since each row in  $\mathbf{X}$  represents a data example, the softmax operation itself can be computed *row-wise*: for each row of  $\mathbf{O}$ , exponentiate all entries and then normalize them by the sum
- triggering *broadcasting* during the summation  $\mathbf{X}\mathbf{W} + \mathbf{b}$ , both the logits  $\mathbf{O} \in \mathbb{R}^{n \times q}$  and output probabilities  $\hat{\mathbf{Y}} \in \mathbb{R}^{n \times q}$  are  $n \times q$  matrices

## 2.2 Multilayer perceptrons

- now, we will introduce our first truly *deep* network
- the simplest deep networks are called *multilayer perceptrons*, and they consist of multiple layers of neurons, each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence)
- we have described the affine transformation, which is a linear transformation added by a bias, in Chapter 1
- consider the model architecture corresponding to our softmax regression example, illustrated in Figure 3

## 2.2 Multilayer perceptrons

- this model mapped our inputs directly to our outputs via a single affine transformation, followed by a softmax operation
- if our labels truly were related to our input data by an affine transformation, then this approach would be sufficient
- but linearity in affine transformations is a *strong assumption*
- for example, linearity implies the weaker assumption of monotonicity: that any increase in our feature must either always cause an increase in our model's output (if the corresponding weight is positive), or always cause a decrease in our model's output (if the corresponding weight is negative); sometimes that makes sense

## 2.2 Multilayer perceptrons

- but what about classifying images of cats and dogs?
- should increasing the intensity of the pixel at location (13, 17) always increase (or always decrease) the likelihood that the image depicts a dog?
- reliance on a linear model corresponds to the implicit assumption that the only requirement for differentiating cats vs. dogs is to assess the brightness of individual pixels
- this approach is doomed to fail in a world where inverting an image preserves the category

## 2.2 Multilayer perceptrons

- and yet, despite the apparent absurdity of linearity here, it is not obvious that we could address the problem with a simple preprocessing fix
- that is because the significance of any pixel depends in complex ways on its context (the values of the surrounding pixels)
- while there might exist a representation of our data that would take into account the relevant interactions among our features, on top of which a linear model would be suitable, we simply do not know how to calculate it by hand
- with deep neural networks, we use training data to jointly learn both a representation via *hidden layers* and a *linear predictor* that acts upon that representation

## 2.2 Multilayer perceptrons

- we can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers
- the easiest way to do this is to stack many fully-connected layers on top of each other
- each layer feeds into the layer above it, until we generate outputs
- we can think of the first  $L - 1$  layers as our *representation* and the final layer as our *linear predictor*, which can be a linear regressor (for regression) or a logistic/softmax regressor (for classification)
- this architecture is commonly called a *multilayer perceptron*, often abbreviated as MLP
- below, we depict the diagram of an MLP (Figure 4)

## 2.2 Multilayer perceptrons

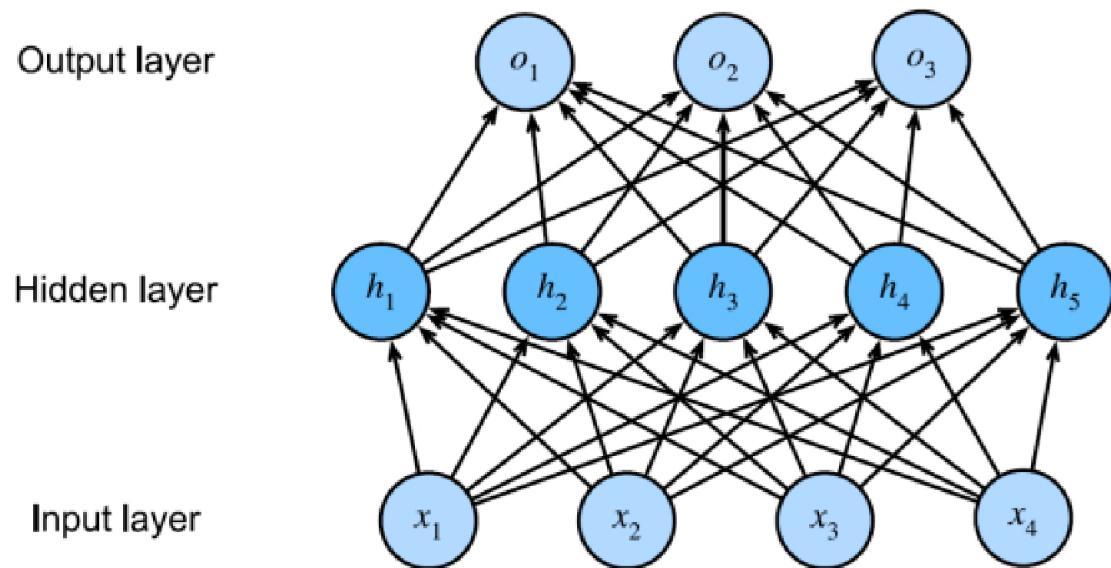


Figure 4: An MLP with a hidden layer of 5 hidden units.

## 2.2 Multilayer perceptrons

- this MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units
- since the input layer does not involve any calculations, producing outputs with this network requires implementing the computations for both the hidden and output layers; thus, the number of layers in this MLP is 2
- note that these layers are both *fully connected*
- every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer

## 2.2 Multilayer perceptrons

- as before, by the matrix  $\mathbf{X} \in \mathbb{R}^{n \times d}$ , we denote a design matrix of  $n$  examples, where each example has  $d$  inputs (features)
- for a one-hidden-layer MLP whose hidden layer has  $h$  hidden units, denote by  $\mathbf{H} \in \mathbb{R}^{n \times h}$  the outputs of the hidden layer, which are *hidden representations*
- in mathematics or code,  $\mathbf{H}$  is also known as a *hidden-layer variable* or a *hidden variable*
- since the hidden and output layers are both fully connected, we have hidden-layer weights  $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$  and biases  $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$  and output-layer weights  $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$  and biases  $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$
- formally, we calculate the outputs  $\mathbf{O} \in \mathbb{R}^{n \times q}$  of the one-hidden-layer MLP as follows:

$$\begin{aligned}\mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned}$$

## 2.2 Multilayer perceptrons

- note that after adding the hidden layer, our model now requires us to track and update additional sets of parameters
- so what have we gained in exchange?
- we might be surprised to find out that – in the model defined above – *we gain nothing for our troubles*
- the reason is plain: the hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units
- an affine function of an affine function is itself an affine function
- moreover, our linear model was already capable of representing any affine function

## 2.2 Multilayer perceptrons

- we can view the equivalence formally, by proving that, for any values of the weights, we can just collapse out the hidden layer, yielding an equivalent single-layer model with parameters  $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$  and  $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$ :

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}.$$

- in order to realize the potential of multilayer architectures, we need one more key ingredient: a *nonlinear activation function*  $\sigma$  to be applied to each hidden unit, following the affine transformation
- the outputs of activation functions (e.g.,  $\sigma(\cdot)$ ) are called *activations*
- in general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned}$$

## 2.2 Multilayer perceptrons

- since each row in  $\mathbf{X}$  corresponds to an example, with some abuse of notation, we define the *nonlinearity*  $\sigma$  to apply to its inputs in a row-wise fashion, i.e., one example at a time
- note that we used the notation for softmax in the same way to denote a row-wise operation before
- often, the activation functions that we apply to hidden layers are not merely row-wise, but element-wise
- that means that, after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units; this is true for most activation functions
- to build more general MLPs, we can continue stacking such hidden layers, e.g.,  $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$  and  $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$ , one on top of the other, yielding ever more expressive models

## 2.2 Multilayer perceptrons

- MLPs can capture complex interactions among our inputs via their hidden neurons, which depend on the values of each of the inputs
- we can easily design hidden nodes to perform arbitrary computation, for instance, basic logic operations on a pair of inputs
- moreover, for certain choices of the activation function, it is widely known that MLPs are *universal approximators*
- even with a single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function, though actually learning that function is the hard part
- we might think of our neural network as being a bit like the C programming language
- the language, like any other modern language, is capable of expressing *any* computable program
- but actually coming up with a program that meets our specifications is the hard part

## 2.2 Multilayer perceptrons

- moreover, just because a single-hidden-layer network *can* learn any function, does not mean that we should try to solve all of our problems with single-hidden-layer networks
- in fact, we can approximate many functions much more compactly by using deeper (vs. wider) networks
- activation functions decide whether a neuron should be activated or not, by calculating the weighted sum and further adding bias to it
- they are differentiable operators to transform input signals to outputs, while most of them add a nonlinearity
- because activation functions are fundamental to deep learning, let us briefly survey some common activation functions

## 2.2 Multilayer perceptrons

- the most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (ReLU)
- ReLU provides a very simple nonlinear transformation
- given an element  $x$ , the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max(x, 0).$$

- informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0
- to gain some intuition, we plot the function in Figure 5; as we can see, the activation function is *piecewise linear*

## 2.2 Multilayer perceptrons

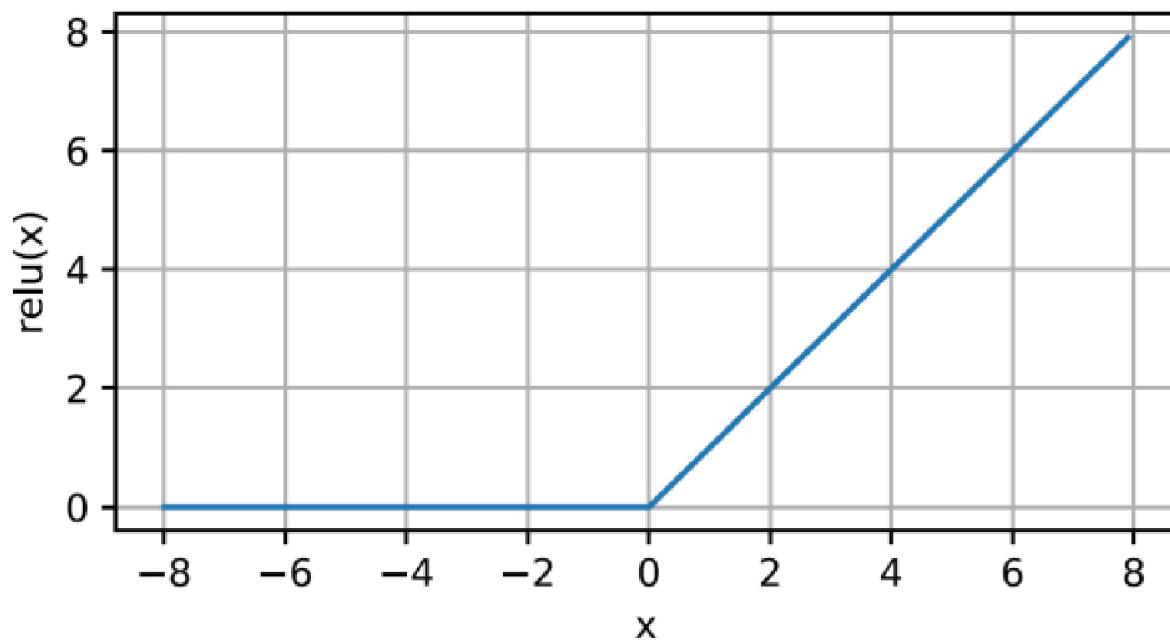


Figure 5: The ReLU function.

## 2.2 Multilayer perceptrons

- when the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1
- note that the ReLU function is not differentiable when the input takes value precisely equal to 0
- in these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0
- we can get away with this, because the input may never actually be zero
- we plot the derivative of the ReLU function in Figure 6

## 2.2 Multilayer perceptrons

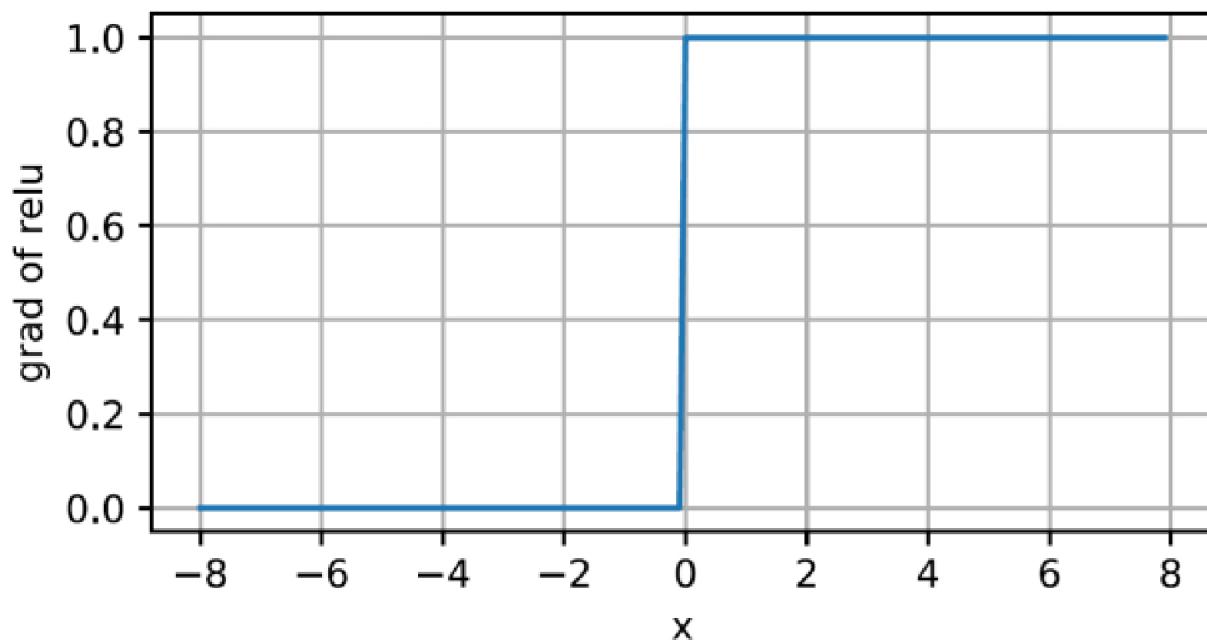


Figure 6: The derivative of the ReLU function.

## 2.2 Multilayer perceptrons

- the reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through
- this makes optimization better behaved and it mitigates the well-documented problem of vanishing gradients that plagued previous versions of neural networks (more on this later)
- note that there are many variants of the ReLU function, including the *parameterized ReLU* (pReLU) function
- this variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

## 2.2 Multilayer perceptrons

- the *sigmoid function* transforms its inputs, for which values lie in the domain  $\mathbb{R}$ , to outputs that lie in the interval  $(0, 1)$
- for that reason, the sigmoid is often called a *squashing function*: it squashes any input in the range  $(-\infty, \infty)$  to some value in the range  $(0, 1)$ :

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

- in the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*
- thus, the pioneers of this field, going all the way back to McCulloch and Pitts, the inventors of the artificial neuron, focused on *thresholding units*
- a thresholding activation takes value 0 when its input is below some threshold, and value 1 when the input exceeds the threshold

## 2.2 Multilayer perceptrons

- when attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit
- sigmoids are still widely used as activation functions on the output units, when we want to interpret the outputs as probabilities for *binary classification* problems (we can think of the sigmoid as a special case of the softmax)
- however, the sigmoid has mostly been replaced by the simpler and more easily trainable ReLU for most use in hidden layers
- in the chapter on recurrent neural networks, we will describe architectures that leverage sigmoid units to control the flow of information across time
- we plot the sigmoid function in Figure 7; note that, when the input is close to 0, the sigmoid function approaches a linear transformation

## 2.2 Multilayer perceptrons

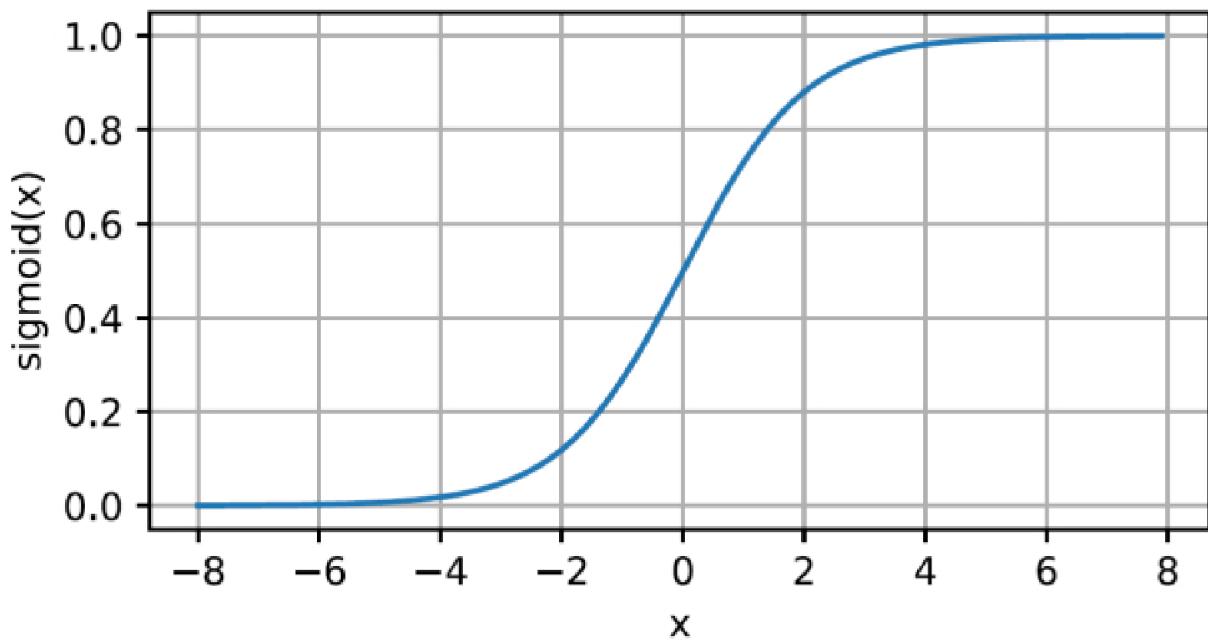


Figure 7: The sigmoid function.

## 2.2 Multilayer perceptrons

- the derivative of the sigmoid function is given by the following equation:

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

- the derivative of the sigmoid function is plotted in Figure 8
- note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25
- as the input diverges from 0 in either direction, the derivative approaches 0

## 2.2 Multilayer perceptrons

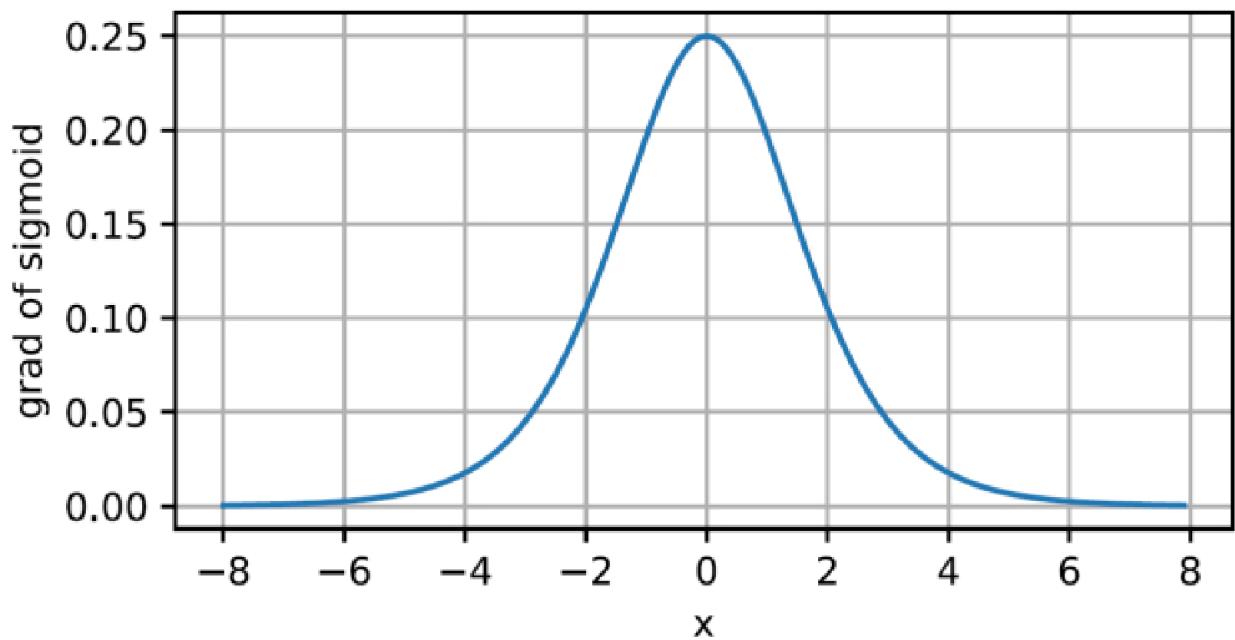


Figure 8: The derivative of the sigmoid function.

## 2.2 Multilayer perceptrons

- like the sigmoid function, the *tanh (hyperbolic tangent)* function also squashes its inputs, transforming them into elements in the interval between  $-1$  and  $1$ :

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

- we plot the tanh function in Figure 9
- note that, as the input nears  $0$ , the tanh function approaches a linear transformation
- although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system

## 2.2 Multilayer perceptrons

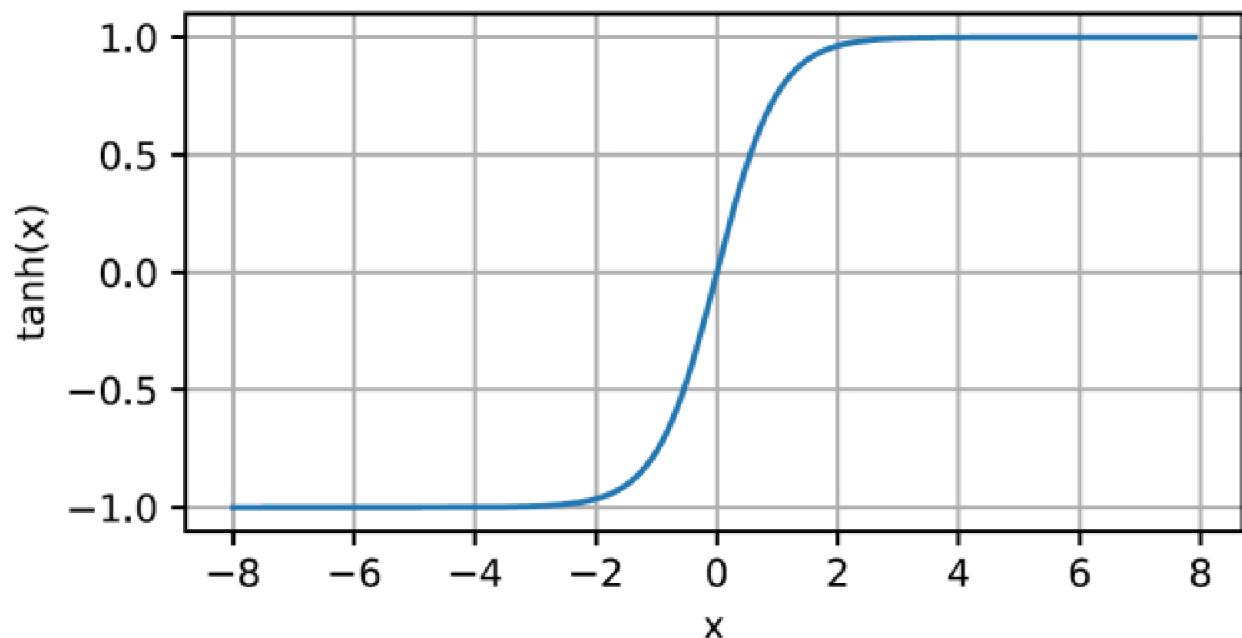


Figure 9: The tanh function.

## 2.2 Multilayer perceptrons

- the derivative of the tanh function is:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x).$$

- the derivative of tanh function is plotted in Figure 10
- as the input nears 0, the derivative of the tanh function approaches a maximum of 1
- and, as we saw with the sigmoid function, as the input moves away from 0 in either direction, the derivative of the tanh function approaches 0

## 2.2 Multilayer perceptrons

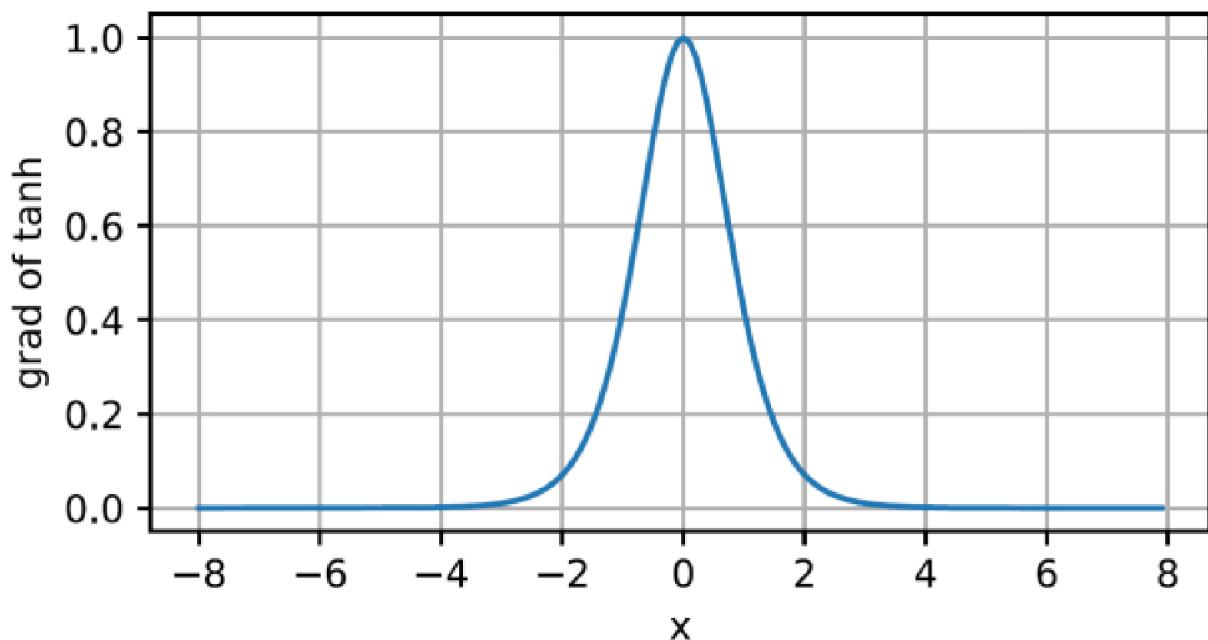


Figure 10: The derivative of the tanh function.

## 2.2 Multilayer perceptrons

- in summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures
- as a side note, our knowledge already puts us in command of a similar toolkit to a practitioner circa 1990
- in some ways, we have an advantage over anyone working in the 1990s, because we can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code
- previously, training these networks required researchers to code up thousands of lines of C and Fortran

## 2.3 Model selection, underfitting, and overfitting

- in machine learning, the goal is to discover *patterns*
- but how can we be sure that we have truly discovered a *general* pattern and not simply memorized our data?
- our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn
- if we are successful in this endeavor, then we could successfully assess the performance even for samples that we have never encountered before
- this problem – how to discover patterns that *generalize* – is the fundamental problem of machine learning

## 2.3 Model selection, underfitting, and overfitting

- the danger is that, when we train models, we access just a small sample of data
- the largest public image datasets contain roughly one million images
- more often, we must learn from only thousands or tens of thousands of data examples
- when working with finite samples, we run the risk that we might discover apparent associations that turn out not to hold up when we collect more data
- the phenomenon of fitting our training data more closely than we fit the underlying distribution is *overfitting*, and the techniques used to combat overfitting are *regularization* techniques

## 2.3 Model selection, underfitting, and overfitting

- in order to discuss this phenomenon more formally, we need to differentiate between training error and generalization error
- the *training error* is the error of our model as calculated on the training dataset, while *generalization error* is the expectation of our model's error were we to apply it to an infinite stream of additional data examples drawn from the same underlying data distribution as our original sample
- problematically, we can never calculate the generalization error exactly
- that is because the stream of infinite data is an imaginary object
- in practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of data examples that were withheld from our training set

## 2.3 Model selection, underfitting, and overfitting

- in the standard supervised learning setting, we assume that both the training data and the test data are drawn *independently* from *identical* distributions
- this is commonly called the *i.i.d. assumption*, which means that the process that samples our data has no memory
- in other words, the second example drawn and the third drawn are no more correlated than the second and the two-millionth sample drawn
- there are common cases where this assumption fails; however, we will assume that the i.i.d. assumption holds

## 2.3 Model selection, underfitting, and overfitting

- when we train our models, we attempt to search for a function that fits the training data as well as possible
- if the function is so flexible that it can catch on to spurious patterns just as easily as to true associations, then it might perform *too well*, without producing a model that generalizes well to unseen data
- this is precisely what we want to avoid or at least control
- many of the techniques in deep learning are heuristics and tricks aimed at guarding against overfitting

## 2.3 Model selection, underfitting, and overfitting

- when we have simple models and abundant data, we expect the generalization error to resemble the training error
- when we work with more complex models and fewer examples, we expect the training error to go down, but the *generalization gap* to grow
- what precisely constitutes model complexity is a complex matter
- many factors govern whether a model will generalize well
- for example, a model with *more parameters* might be considered more complex
- a model whose parameters can take a *wider range of values* might be more complex
- often with neural networks, we think of a model that takes *more training iterations* as more complex, and one subject to *early stopping* (fewer training iterations) as less complex

## 2.3 Model selection, underfitting, and overfitting

- it can be difficult to compare the complexity among members of substantially different model classes (say, decision trees vs. neural networks)
- for now, a simple rule of thumb is quite useful: a model that can readily explain arbitrary facts is what can be viewed as complex, whereas one that has only a limited expressive power, but still manages to explain the data well, is probably closer to the truth
- this conforms to the principle of Occam's razor: given two explanations for something, the explanation most likely to be correct is the simplest one – the one that makes fewer assumptions

## 2.3 Model selection, underfitting, and overfitting

- next, we will focus on a few factors that tend to influence the generalizability of a model class:
  - ➊ The number of tunable parameters. When the number of tunable parameters, sometimes called the *degrees of freedom*, is large, models tend to be more susceptible to overfitting.
  - ➋ The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to overfitting.
  - ➌ The number of training examples. It is trivially easy to overfit a dataset containing only one or two examples, even if our model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

## 2.3 Model selection, underfitting, and overfitting

- in machine learning, we usually select our final model after evaluating several candidate models; this process is called *model selection*
- sometimes, the models subject to comparison are fundamentally different in nature (say, decision trees vs. linear models)
- at other times, we are comparing members of the same class of models that have been trained with *different hyperparameter settings*
- with MLPs, for example, we may wish to compare models with different numbers of hidden layers, different numbers of hidden units, and various choices of the activation functions applied to each hidden layer
- in order to determine the best among our candidate models, we will typically employ a validation dataset

## 2.3 Model selection, underfitting, and overfitting

- in principle, we should not touch our test set until after we have chosen all our hyperparameters
- were we to use the test data in the model selection process, there is a risk that we might *overfit the test data*, which is very problematic
- if we overfit our training data, there is always the evaluation on test data to keep us honest
- but if we overfit the test data, how would we ever know?
- thus, we should never rely on the test data for model selection
- and yet we cannot rely solely on the training data for model selection either, because we cannot estimate the generalization error on the very data that we use to train the model

## 2.3 Model selection, underfitting, and overfitting

- the common practice to address this problem is to split our data *three ways*, incorporating a *validation set*, in addition to the training and test sets
- the performance on this validation set will be used for model selection
- when training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set
- one popular solution to this problem is to use *K-fold cross-validation*
- here, the original training data is split into  $K$  non-overlapping subsets
- then, model training and validation are executed  $K$  times, each time training on  $K - 1$  subsets and validating on a different subset (the one not used for training in that round)
- finally, the training and validation errors are estimated by averaging over the results from the  $K$  experiments

## 2.3 Model selection, underfitting, and overfitting

- when we compare the training and validation errors, we should be aware of two common situations
- first, we should watch out for cases when our training error and validation error are both substantial, but there is a *little gap* between them
- if the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model
- moreover, since the *generalization gap* between our training and validation errors is small, we have reason to believe that we could use a more complex model
- this phenomenon is known as *underfitting*

## 2.3 Model selection, underfitting, and overfitting

- on the other hand, as we discussed above, we should watch out for the cases when our training error is significantly lower than our validation error, indicating severe *overfitting*
- note that overfitting is not always a bad thing
- with deep learning especially, it is well known that the best predictive models often perform far better on training data than on holdout data
- ultimately, we usually care more about the validation error than about the gap between the training and validation errors

## 2.3 Model selection, underfitting, and overfitting

- whether we overfit or underfit can depend both on the complexity of our model and the size of the available training datasets, two topics that we discuss below
- to illustrate some classical intuition about overfitting and model complexity, we give an example using polynomials
- given training data consisting of a single feature  $x$  and a corresponding real-valued label  $y$ , we try to find the polynomial of degree  $d$  to estimate the labels  $y$

## 2.3 Model selection, underfitting, and overfitting

- a higher-order polynomial function is more complex than a lower-order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider
- fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower degree polynomials
- in fact, whenever the data examples each have a distinct value of  $x$ , a polynomial function with degree equal to the number of data examples can fit the training set perfectly
- we visualize the relationship between polynomial degree and underfitting vs. overfitting in Figure 11

## 2.3 Model selection, underfitting, and overfitting

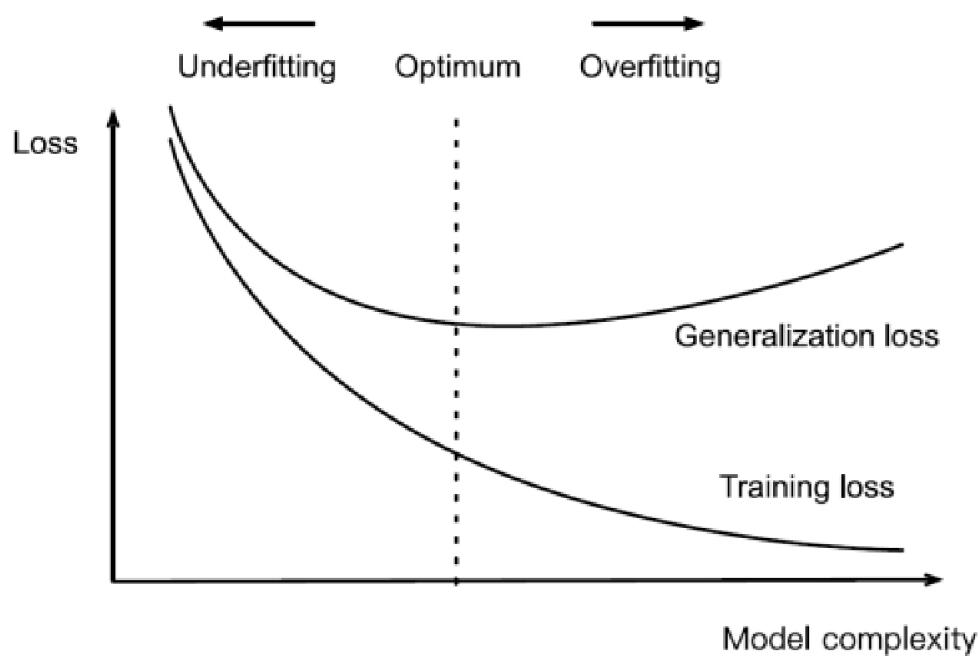


Figure 11: Influence of model complexity on underfitting and overfitting.

## 2.3 Model selection, underfitting, and overfitting

- the other big consideration to bear in mind is the *dataset size*
- fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting
- as we increase the amount of training data, the generalization error typically decreases
- moreover, in general, more data never hurts
- for a fixed task and data distribution, there is typically a relationship between model complexity and dataset size

## 2.3 Model selection, underfitting, and overfitting

- given more data, we might profitably attempt to fit a more complex model
- without sufficient data, simpler models may be more difficult to beat
- for many tasks, deep learning only outperforms linear models when many thousands of training examples are available
- in part, the current success of deep learning owes to the current abundance of massive datasets due to Internet companies, cheap storage, connected devices, and the broad digitization of the economy

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 4

October 25th, 2022

## 2.4 Weight decay

- now that we have characterized the problem of overfitting, we can introduce some standard techniques for regularizing models
- recall that we can always mitigate overfitting by going out and collecting more training data
- that can be costly, time consuming, or entirely out of our control, making it impossible in the short run
- for now, we can assume that we already have as much high-quality data as our resources permit and focus on regularization techniques

## 2.4 Weight decay

- *weight decay* (generally called  $\ell_2$  regularization), might be the most widely-used technique for regularizing parametric machine learning models
- the technique is motivated by the basic intuition that the *simplest* models are the ones for which some norm of their weight vector is as small as possible
- the most common method for ensuring a small weight vector is to add its norm as a penalty term to the problem of minimizing the loss
- thus, we replace our original objective, *minimizing the prediction loss on the training labels*, with the new objective, *minimizing the sum of the prediction loss and the penalty term*
- now, if our weight vector grows too large, our learning algorithm might focus on minimizing the weight norm vs. minimizing the training error, which is exactly what we want

## 2.4 Weight decay

- for example, consider the mean squared error loss given by:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2.$$

- to penalize the size of the weight vector, we must somehow add the weight norm to the loss function, but how should the model trade off the standard loss for this new additive penalty?
- in practice, we characterize this trade-off via the *regularization constant*  $\lambda$ , a nonnegative hyperparameter that we fit using validation data
- for the one-hidden-layer MLP discussed above, the new objective becomes:

$$\mathcal{L}(\mathbf{w}) + \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right),$$

where the Frobenius norm is a generalization for matrices of the Euclidean  $\ell_2$  norm, and is defined as:  $\|\mathbf{W}\|_F = \sqrt{\text{Tr}(\mathbf{W}^\top \mathbf{W})}$

## 2.4 Weight decay

- for  $\lambda = 0$ , we recover our original loss function
- for  $\lambda > 0$ , we restrict the size of  $\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2$ , the norm of the weights of our network
- we divide by 2 by convention: when we take the derivative of a quadratic function, the 2 and 1/2 cancel out, ensuring that the expression for the update looks nice and simple
- we might wonder why we work with the squared norm and not the standard norm (i.e., the Euclidean  $\ell_2$  norm); we do this for computational convenience
- by squaring the  $\ell_2$  norm, we remove the square root, leaving the sum of squares of each component of the weight vector
- this makes the derivative of the penalty easy to compute: the sum of derivatives equals the derivative of the sum

## 2.4 Weight decay

- now, when applying the mini-batch stochastic gradient descent algorithm, in addition to updating the weights based on the amount by which the prediction differs from the actual label, we also shrink the size of the weights  $\mathbf{W}^{(1)}$  and  $\mathbf{W}^{(2)}$  towards zero
- that is why the method is sometimes called “weight decay”: given the penalty term alone, our optimization algorithm *decays* the weight at each step of training
- smaller values of  $\lambda$  correspond to less constrained weights, whereas larger values of  $\lambda$  constrain the weights more considerably

- we introduced the classical approach to regularizing statistical models by penalizing the  $\ell_2$  norm of the weights
- in probabilistic terms, we justified this technique by arguing that we have assumed a prior belief that weights take values from a Gaussian distribution with mean zero
- more intuitively, we might argue that we encouraged the model to spread out its weights among many features, rather than depending too much on a small number of potentially false associations

## 2.5 Dropout

- faced with more features than examples, linear models tend to overfit
- but given more examples than features, we can generally count on linear models not to overfit
- unfortunately, the reliability with which linear models generalize comes at a cost
- naively applied, linear models do not take into account interactions among features
- for every feature, a linear model must assign either a positive or a negative weight, ignoring context

## 2.5 Dropout

- this fundamental tension between generalizability and flexibility was described as the *bias-variance trade-off*
- linear models have high bias: they can only represent a small class of functions
- however, these models have low variance: they give similar results across different random samples of the data
- deep neural networks inhabit the opposite end of the bias-variance spectrum
- unlike linear models, neural networks are not confined to looking at each feature individually
- they can learn interactions among groups of features

## 2.5 Dropout

- even when we have far more examples than features, deep neural networks are capable of overfitting
- in 2017, a group of researchers demonstrated the extreme flexibility of neural networks by training deep nets on randomly-labeled images
- despite the absence of any true pattern linking the inputs to the outputs, they found that the neural network optimized by stochastic gradient descent could label every image in the training set perfectly
- consider what this means: if the labels are assigned uniformly at random and there are 10 classes, then no classifier can do better than 10% accuracy on holdout data
- the generalization gap here is a whopping 90%
- if our models are so expressive that they can overfit this badly, then when should we expect them not to overfit?

## 2.5 Dropout

- let us think briefly about what we expect from a good predictive model
- we want it to perform well on unseen data
- classical generalization theory suggests that, to close the gap between train and test performance, we should aim for a *simple* model
- simplicity can come in the form of a small number of dimensions
- additionally, as we saw when discussing weight decay ( $\ell_2$  regularization), the norm of the weights also represents a useful measure of simplicity
- another useful notion of simplicity is *smoothness*, i.e., that the function should not be sensitive to noise
- for instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless

## 2.5 Dropout

- thus, we could inject noise into each layer of the network, before calculating the subsequent layer during training
- when training a deep network with many layers, injecting noise enforces smoothness on the input-output mapping
- this idea, called *dropout*, involves injecting noise while computing each internal layer during forward propagation, and it has become a standard technique for training neural networks
- the method is called *dropout* because we literally *drop out* some neurons during training
- throughout training, on each iteration, standard dropout consists of zeroing out some fraction of the nodes in each layer, before calculating the subsequent layer

## 2.5 Dropout

- we may argue that neural network overfitting is characterized by a state in which each layer relies on a specific pattern of activations in the previous layer, this condition being called *co-adaptation*
- dropout breaks up co-adaptation, by not allowing neurons to be too reliant on each other
- in standard dropout regularization, with *dropout probability*  $p$ , each intermediate activation  $h$  is replaced by a random variable  $h'$  as follows:

$$h' = \begin{cases} 0, & \text{with probability } p \\ \frac{h}{1-p}, & \text{otherwise} \end{cases}.$$

- thus, each layer is normalized by the fraction of nodes that were retained (not dropped out), and so the expectation remains unchanged, i.e.,  $E[h'] = h$

## 2.5 Dropout

- recall the MLP with a hidden layer and 5 hidden units in Figure 4
- when we apply dropout to a hidden layer, zeroing out each hidden unit with probability  $p$ , the result can be viewed as a network containing only a subset of the original neurons
- in Figure 12,  $h_2$  and  $h_5$  are removed
- consequently, the calculation of the outputs no longer depends on  $h_2$  or  $h_5$ , and their respective gradient also vanishes when performing backpropagation
- in this way, the calculation of the output layer cannot be overly dependent on any one element of  $h_1, \dots, h_5$

## 2.5 Dropout

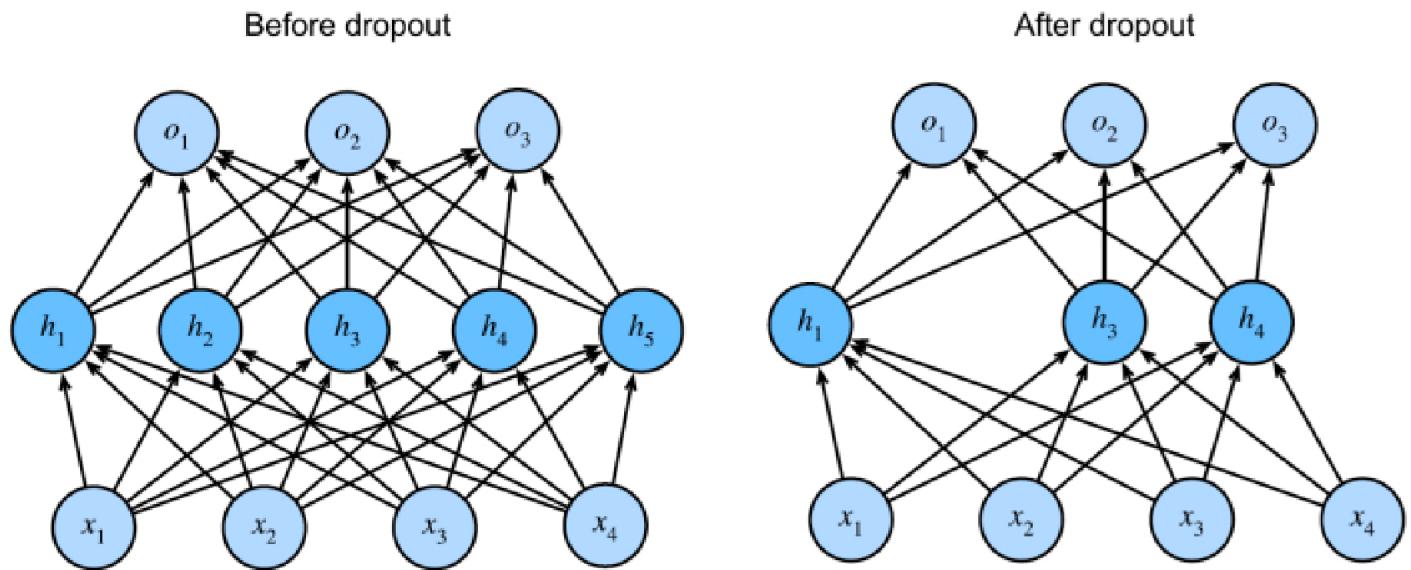


Figure 12: MLP before and after dropout.

## 2.5 Dropout

- dropout is disabled at test time: given a trained model and a new example, we do not drop out any nodes, and thus do not need to normalize
- to implement the dropout function for a single layer, we must draw as many samples from a Bernoulli (binary) random variable as our layer has dimensions, where the random variable takes value 1 (keep) with probability  $1 - p$  and 0 (drop) with probability  $p$
- one easy way to implement this is to first draw samples from the uniform distribution  $\mathcal{U}(0, 1)$
- then we can keep those nodes for which the corresponding sample is greater than  $p$ , dropping the rest
- thus, we drop out the elements in the layer with probability  $p$ , rescaling the remainder as described above: dividing the survivors by  $1 - p$

## 2.6 Forward propagation, backward propagation, and computational graphs

- when using modern deep learning frameworks, we only have to define the calculations involved in *forward propagation* through the model
- when it comes to calculating the gradients in order to perform gradient descent, we just invoke the backpropagation function provided by the deep learning framework
- the automatic calculation of gradients (*automatic differentiation*) profoundly simplifies the implementation of deep learning algorithms
- before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand
- surprisingly often, academic papers had to allocate numerous pages to deriving update rules
- while we must continue to rely on automatic differentiation so we can focus on the interesting parts, we should know how these gradients are calculated under the hood, if we want to go beyond a shallow understanding of deep learning

## 2.6 Forward propagation, backward propagation, and computational graphs

- in this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*)
- to convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs
- to start, we focus our exposition on a one-hidden-layer MLP with weight decay ( $\ell_2$  regularization)
- *forward propagation* (or *forward pass*) refers to the calculation and storage of *intermediate variables* (including outputs) for a neural network in order, from the input layer to the output layer

## 2.6 Forward propagation, backward propagation, and computational graphs

- for the sake of simplicity, let us assume that the input example is  $\mathbf{x} \in \mathbb{R}^d$  and that our hidden layer does not include a bias term
- here, the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)} \mathbf{x},$$

where  $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  is the weight parameter of the hidden layer

- after running the intermediate variable  $\mathbf{z} \in \mathbb{R}^h$  through the activation function  $\phi$ , we obtain our hidden activation vector of length  $h$ :

$$\mathbf{h} = \phi(\mathbf{z}).$$

- the hidden variable  $\mathbf{h} \in \mathbb{R}^h$  is also an intermediate variable
- assuming that the parameters of the output layer only possess a weight of  $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ , we can obtain an output vector of length  $q$ :

$$\mathbf{o} = \mathbf{W}^{(2)} \mathbf{h}.$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- assuming that the loss function is  $l$ , the squared error loss, and the example label is  $\mathbf{y} \in \mathbb{R}^q$ , we can then calculate the loss term for a single data example as:

$$L = l(\mathbf{o}, \mathbf{y}) = \frac{1}{2}(\mathbf{o} - \mathbf{y})^\top (\mathbf{o} - \mathbf{y}).$$

- according to the definition of  $\ell_2$  regularization, given the hyperparameter  $\lambda$ , the regularization term is:

$$s = \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right), \quad (1)$$

where the Frobenius norm is defined as:  $\|\mathbf{W}\|_F = \sqrt{\text{Tr}(\mathbf{W}^\top \mathbf{W})}$

- finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

- we refer to  $J$  as the *objective function* in the following discussion

## 2.6 Forward propagation, backward propagation, and computational graphs

- plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation
- Figure 13 contains the graph associated with the simple network described above, where squares denote variables and circles denote operators
- the lower-left corner signifies the input and the upper-right corner is the output
- notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward

## 2.6 Forward propagation, backward propagation, and computational graphs

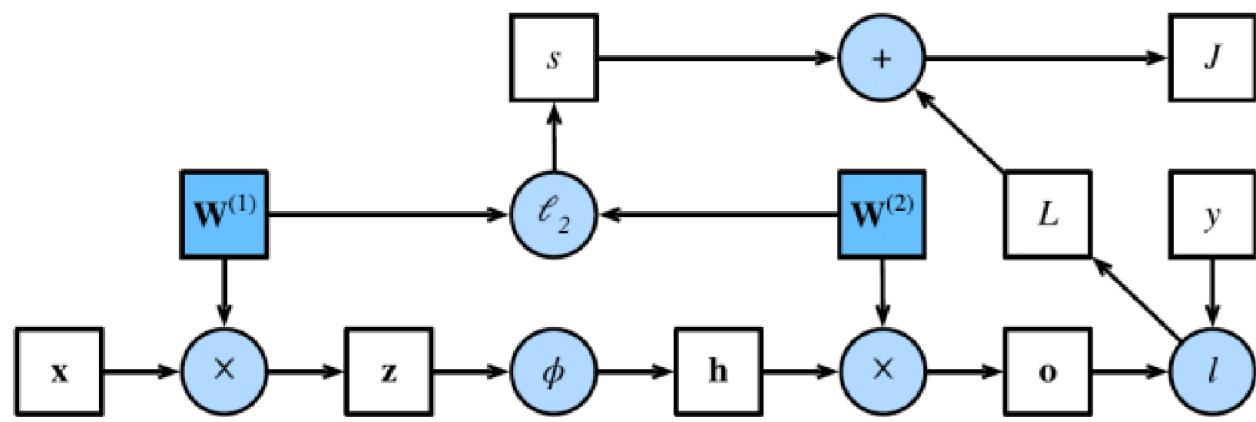


Figure 13: Computational graph of forward propagation.

## 2.6 Forward propagation, backward propagation, and computational graphs

- *backpropagation* refers to the method of calculating the gradient of neural network parameters
- in short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus
- the algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters
- assume that we have functions  $Y = f(X)$  and  $Z = g(Y)$ , in which the input and the output  $X, Y, Z$  are scalars, vectors, or matrices of arbitrary shapes

## 2.6 Forward propagation, backward propagation, and computational graphs

- by using the chain rule, we can compute the derivative of Z with respect to X via the chain rule as:

$$\frac{\partial Z}{\partial X} = \text{prod} \left( \frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right).$$

- here, we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out
- for vectors, this is straightforward: it is simply matrix-matrix multiplication
- for matrices, we use the appropriate counterpart
- the operator prod hides all the notation overhead

## 2.6 Forward propagation, backward propagation, and computational graphs

- recall that the parameters of the simple network with one hidden layer, whose computational graph is in Figure 13, are  $\mathbf{W}^{(1)}$  and  $\mathbf{W}^{(2)}$
- the objective of backpropagation is to calculate the gradients  $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$  and  $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$
- to accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter
- the order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters

## 2.6 Forward propagation, backward propagation, and computational graphs

- we start by calculating the gradient  $\frac{\partial J}{\partial \mathbf{W}^{(2)}} \in \mathbb{R}^{q \times h}$  of the model parameters closest to the output layer
- first, we can write that:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \frac{\partial(L + s)}{\partial \mathbf{W}^{(2)}} \\ &= \frac{\partial L}{\partial \mathbf{W}^{(2)}} + \frac{\partial s}{\partial \mathbf{W}^{(2)}}.\end{aligned}\tag{2}$$

- using the chain rule for the first term in (2) yields:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(2)}} &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) \\ &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}}, \frac{\partial(\mathbf{W}^{(2)} \mathbf{h})}{\partial \mathbf{W}^{(2)}} \right) \\ &= \frac{\partial L}{\partial \mathbf{o}} \mathbf{h}^\top.\end{aligned}\tag{3}$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- now, we just have to compute the gradient of the loss function  $L$  with respect to the variable of the output layer  $\mathbf{o}$ :

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{o}} &= \frac{\partial}{\partial \mathbf{o}} \left( \frac{1}{2} (\mathbf{o} - \mathbf{y})^\top (\mathbf{o} - \mathbf{y}) \right) \\ &= \mathbf{o} - \mathbf{y}.\end{aligned}$$

- thus, (3) becomes:

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = (\mathbf{o} - \mathbf{y}) \mathbf{h}^\top.$$

- we next calculate the gradient of the regularization term  $s$  with respect to  $\mathbf{W}^{(2)}$ :

$$\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

- putting it all together, (2) becomes:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = (\mathbf{o} - \mathbf{y}) \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}.$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- in order to obtain the gradient  $\frac{\partial J}{\partial \mathbf{W}^{(1)}} \in \mathbb{R}^{h \times d}$  of the model parameters closest to the input layer, we first write that:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \frac{\partial(L + s)}{\partial \mathbf{W}^{(1)}} \\ &= \frac{\partial L}{\partial \mathbf{W}^{(1)}} + \frac{\partial s}{\partial \mathbf{W}^{(1)}}.\end{aligned}\tag{4}$$

- for the first term in (4), according to the chain rule, we get:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}^{(1)}} &= \text{prod}\left(\frac{\partial L}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) \\ &= \text{prod}\left(\frac{\partial L}{\partial \mathbf{z}}, \frac{\partial(\mathbf{W}^{(1)}\mathbf{x})}{\partial \mathbf{W}^{(1)}}\right) \\ &= \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^\top.\end{aligned}$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- since the activation function  $\phi$  applies element-wise, calculating the gradient  $\frac{\partial L}{\partial \mathbf{z}} \in \mathbb{R}^h$  of the intermediate variable  $\mathbf{z}$  requires that we use the element-wise matrix multiplication operator, which we denote by  $\odot$ :

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{z}} &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) \\ &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{h}}, \frac{\partial \phi(\mathbf{z})}{\partial \mathbf{z}} \right) \\ &= \frac{\partial L}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}).\end{aligned}\tag{5}$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- the gradient with respect to the hidden layer's outputs  $\frac{\partial L}{\partial \mathbf{h}} \in \mathbb{R}^h$  is given by the chain rule:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{h}} &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) \\ &= \text{prod} \left( \frac{\partial L}{\partial \mathbf{o}}, \frac{\partial (\mathbf{W}^{(2)} \mathbf{h})}{\partial \mathbf{h}} \right) \\ &= \mathbf{W}^{(2)\top} \frac{\partial L}{\partial \mathbf{o}} \\ &= \mathbf{W}^{(2)\top} (\mathbf{o} - \mathbf{y}).\end{aligned}$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- now, (5) becomes:

$$\frac{\partial L}{\partial \mathbf{z}} = (\mathbf{W}^{(2)\top}(\mathbf{o} - \mathbf{y})) \odot \phi'(\mathbf{z}).$$

- we similarly as before calculate the gradient of the regularization term  $s$  with respect to  $\mathbf{W}^{(1)}$ :

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}.$$

- putting it all together, (4) is finally given as:

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = ((\mathbf{W}^{(2)\top}(\mathbf{o} - \mathbf{y})) \odot \phi'(\mathbf{z})) \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

## 2.6 Forward propagation, backward propagation, and computational graphs

- thus, we take the following steps to compute  $\frac{\partial J}{\partial \mathbf{W}^{(2)}}$ :

- ① Compute  $\frac{\partial L}{\partial \mathbf{o}} = \mathbf{o} - \mathbf{y}$
- ② Compute  $\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}} \mathbf{h}^T$
- ③ Compute  $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$
- ④ Compute  $\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{W}^{(2)}} + \frac{\partial s}{\partial \mathbf{W}^{(2)}}$

- and the following steps to compute  $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$ :

- ① Compute  $\frac{\partial L}{\partial \mathbf{h}} = \mathbf{W}^{(2)T} \frac{\partial L}{\partial \mathbf{o}}$  ( $\frac{\partial L}{\partial \mathbf{o}}$  is already computed in step 1 above)
- ② Compute  $\frac{\partial L}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$
- ③ Compute  $\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^T$
- ④ Compute  $\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$
- ⑤ Compute  $\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{W}^{(1)}} + \frac{\partial s}{\partial \mathbf{W}^{(1)}}$

## 2.6 Forward propagation, backward propagation, and computational graphs

- when training neural networks, forward and backward propagation depend on each other
- in particular, for forward propagation, we traverse the computational graph in the direction of dependencies, and compute all the variables on its path
- these are then used for backpropagation, where the compute order on the graph is reversed
- take the aforementioned simple network as an example to illustrate
- on one hand, computing the regularization term (1) during forward propagation depends on the current values of model parameters  $\mathbf{W}^{(1)}$  and  $\mathbf{W}^{(2)}$
- they are given by the optimization algorithm according to backpropagation, in the latest iteration
- on the other hand, the gradient calculation for the parameter (3) during backpropagation depends on the current value of the hidden variable  $\mathbf{h}$ , which is given by forward propagation

## 2.6 Forward propagation, backward propagation, and computational graphs

- therefore, when training neural networks, after model parameters are *initialized*, we alternate *forward propagation* with *backpropagation*, *updating* model parameters using *gradients* given by backpropagation
- note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations
- one of the consequences is that we need to retain the intermediate values until backpropagation is complete
- this is also one of the reasons why training requires significantly more memory than plain prediction
- besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size
- thus, training deeper networks using larger batch sizes more easily leads to out of memory errors

## 2.7 Numerical stability and initialization

- thus far, every model that we discussed requires that we initialize its parameters according to some pre-specified distribution
- until now, we took the initialization scheme for granted, and didn't discuss the details of how these choices are made
- we might have even gotten the impression that these choices are not especially important
- to the contrary, the choice of initialization scheme plays a significant role in neural network learning, and it can be crucial for maintaining numerical stability

## 2.7 Numerical stability and initialization

- moreover, these choices can be tied up in interesting ways with the choice of the nonlinear activation function
- which function we choose and how we initialize parameters can determine how quickly our optimization algorithm converges
- poor choices here can cause us to encounter exploding or vanishing gradients while training
- in this section, we delve into these topics with greater detail and discuss some useful heuristics

## 2.7 Numerical stability and initialization

- consider a deep network with  $L$  layers, input  $\mathbf{x}$  and output  $\mathbf{o}$
- with each layer  $l$  defined by a transformation  $f_l$  parameterized by weights  $\mathbf{W}^{(l)}$ , whose hidden variable is  $\mathbf{h}^{(l)}$  (let  $\mathbf{h}^{(0)} = \mathbf{x}$ ), our network can be expressed as:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \dots \circ f_1(\mathbf{x}),$$

where  $\circ$  denotes function composition:  $f \circ g(\mathbf{x}) = f(g(\mathbf{x}))$

- if all the hidden variables and the input are vectors, we can write the gradient of  $\mathbf{o}$  with respect to any set of parameters  $\mathbf{W}^{(l)}$  as follows:

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(l)}} = \underbrace{\frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{h}^{(L-1)}}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \dots \cdot \underbrace{\frac{\partial \mathbf{h}^{(l+1)}}{\partial \mathbf{h}^{(l)}}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \cdot \underbrace{\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

- in other words, this gradient is the product of  $L - I$  matrices  $\mathbf{M}^{(L)} \dots \mathbf{M}^{(I+1)}$  and the gradient vector  $\mathbf{v}^{(I)}$
- thus, we are susceptible to the same problems of numerical underflow that often appear when multiplying together too many probabilities
- when dealing with probabilities, a common trick is to switch into log-space, i.e., shifting pressure from the mantissa to the exponent of the numerical representation
- unfortunately, our problem above is more serious: initially, the matrices  $\mathbf{M}^{(I)}$  may have a wide variety of eigenvalues
- they might be small or large, and their product might be *very large* or *very small*

## 2.7 Numerical stability and initialization

- the risks posed by unstable gradients go beyond numerical representation
- gradients of unpredictable magnitude also threaten the stability of our optimization algorithms
- we may be facing parameter updates that are either:
  - excessively large, destroying our model (the *exploding gradient* problem);
  - excessively small (the *vanishing gradient* problem), rendering learning impossible as parameters hardly move on each update.
- the vanishing gradient problem is often caused by the choice of the activation function that is appended following each layer's linear operations
- historically, the sigmoid function  $1/(1 + \exp(-x))$  was popular, because it resembles a thresholding function
- since early artificial neural networks were inspired by biological neural networks, the idea of neurons that fire either *fully* or *not at all* (like biological neurons) seemed appealing
- let us take a closer look at the sigmoid to see why it can cause vanishing gradients

## 2.7 Numerical stability and initialization

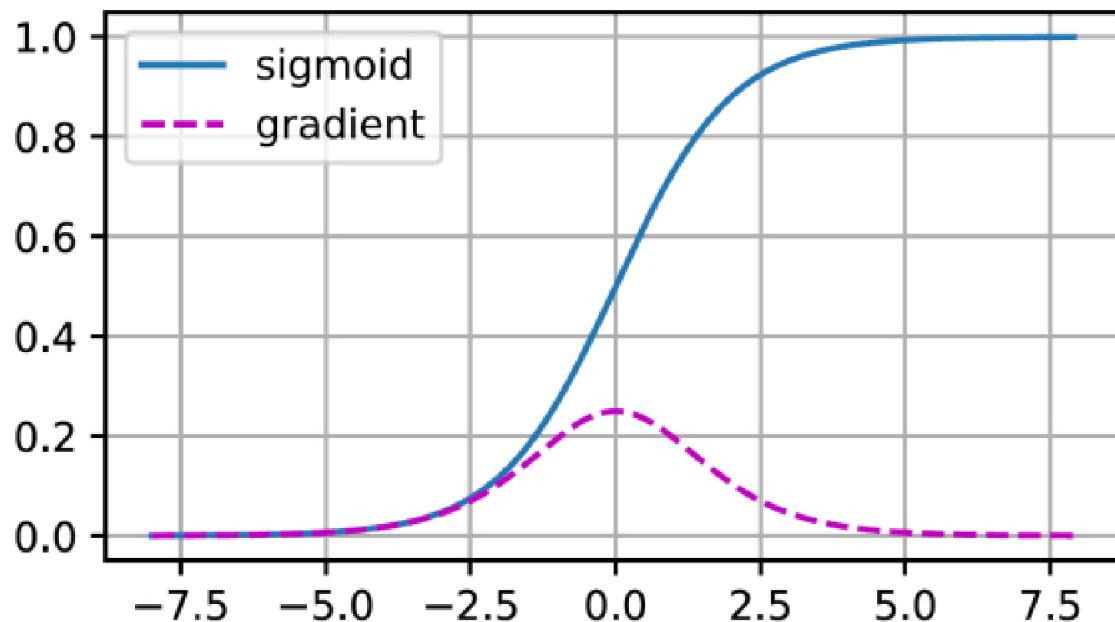


Figure 14: The sigmoid function and its derivative.

## 2.7 Numerical stability and initialization

- as we can see, the sigmoid's derivative vanishes both when its inputs are large and when they are small
- moreover, when backpropagating through many layers (unless we are in the zone where the input to the sigmoid is close to zero), the gradients of the overall product may vanish
- when our network has many layers, unless we are careful, the gradient will likely be cut off at some layer
- indeed, this problem used to plague deep network training
- consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice today

- the opposite problem, when gradients explode, can be similarly troublesome
- to illustrate this a bit better, we can draw 100 Gaussian random matrices and multiply them with some initial matrix
- if we choose the variance  $\sigma^2 = 1$ , we will see that the matrix product explodes
- when this happens due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge

## 2.7 Numerical stability and initialization

- another problem in neural network design is the symmetry inherent in their parametrization
- assume that we have a simple MLP with one hidden layer and two units
- in this case, we could permute the weights  $W^{(1)}$  of the first layer and likewise permute the weights of the output layer to obtain the same function
- there is nothing special differentiating the first hidden unit vs. the second hidden unit
- in other words, we have permutation symmetry among the hidden units of each layer

## 2.7 Numerical stability and initialization

- this is more than just a theoretical problem
- consider a one-hidden-layer MLP with two hidden units
- for illustration, suppose that the output layer transforms the two hidden units into only one output unit
- imagine what would happen if we initialized all of the parameters of the hidden layer as  $\mathbf{W}^{(1)} = c$ , for some constant  $c$
- in this case, during forward propagation, either hidden unit takes the same inputs and parameters, producing the same activation, which is fed to the output unit

- during backpropagation, differentiating the output unit with respect to parameters  $W^{(1)}$  gives a gradient whose elements all take the same value
- thus, after gradient-based iteration (e.g., mini-batch stochastic gradient descent), all the elements of  $W^{(1)}$  still take the same value
- such iterations would never break the symmetry on their own, and we might never be able to realize the network's expressive power
- the hidden layer would behave as if it had only a single unit
- note that, while mini-batch stochastic gradient descent would not break this symmetry, dropout regularization would

## 2.7 Numerical stability and initialization

- one way of addressing – or at least mitigating – the issues raised above is through careful initialization
- additional care during optimization and suitable regularization can further enhance stability
- let us look at the scale distribution of an output (e.g., a hidden variable)  $o_i$  for some fully-connected layer *without nonlinearities*
- with  $n_{\text{in}}$  inputs  $x_j$  and their associated weights  $w_{ij}$  for this layer, an output is given by:

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j.$$

## 2.7 Numerical stability and initialization

- the weights  $w_{ij}$  are all drawn independently from the same distribution
- furthermore, let us assume that this distribution has zero mean and variance  $\sigma^2$
- note that this does not mean that the distribution has to be Gaussian, just that the mean and variance need to exist
- for now, let us assume that the inputs to the layer  $x_i$  also have zero mean and variance  $\gamma^2$ , and that they are independent of  $w_{ij}$  and independent of each other
- in this case, we can compute the mean and variance of  $o_i$  as follows:

## 2.7 Numerical stability and initialization

$$\begin{aligned} E[o_i] &= \sum_{j=1}^{n_{in}} E[w_{ij}x_j] \\ &= \sum_{j=1}^{n_{in}} E[w_{ij}]E[x_j] \\ &= 0, \\ \text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\ &= \sum_{j=1}^{n_{in}} E[w_{ij}^2 x_j^2] - 0 \\ &= \sum_{j=1}^{n_{in}} E[w_{ij}^2]E[x_j^2] \\ &= n_{in}\sigma^2\gamma^2. \end{aligned}$$

## 2.7 Numerical stability and initialization

- one way to keep the variance fixed is to set  $n_{\text{in}}\sigma^2 = 1$ ; now consider backpropagation
- there, we face a similar problem, although with gradients being propagated from the layers closer to the output
- using the same reasoning as for forward propagation, we see that the gradients' variance can explode, unless  $n_{\text{out}}\sigma^2 = 1$ , where  $n_{\text{out}}$  is the number of outputs of this layer
- this leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously
- instead, we simply try to satisfy:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or, equivalently, } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

## 2.7 Numerical stability and initialization

- this is the reasoning underlying the now-standard and practically beneficial *Xavier initialization*, named after the first author of its creators (Xavier Glorot)
- typically, the Xavier initialization samples weights from a Gaussian distribution with zero mean and variance  $\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$
- we can also adapt Xavier's intuition to choose the variance when sampling weights from a uniform distribution
- note that the uniform distribution  $\mathcal{U}(-a, a)$  has variance  $\frac{a^2}{3}$
- plugging  $\frac{a^2}{3}$  into our condition on  $\sigma^2$  yields the suggestion to initialize according to:

$$\mathcal{U}\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right).$$

- though the assumption for nonexistence of nonlinearities in the above mathematical reasoning can be easily violated in neural networks, the Xavier initialization method turns out to work well in practice

## 2.8 Deep learning computation

- alongside giant datasets and powerful hardware, great software tools have played an indispensable role in the rapid progress of deep learning
- starting with the groundbreaking *Theano* library released in 2007, flexible open-source tools have enabled researchers to rapidly prototype models, avoiding repetitive work when recycling standard components, while still maintaining the ability to make low-level modifications
- over time, deep learning's libraries have evolved to offer increasingly coarse abstractions
- just as semiconductor designers went from specifying transistors to logical circuits to writing code, neural networks researchers have moved from thinking about the behavior of individual artificial neurons to conceiving of networks in terms of whole layers, and now often design architectures with far coarser *blocks* in mind

## 2.8 Deep learning computation

- when we introduced neural networks, we focused on linear models with a single output
- here, the entire model consists of just a single neuron
- note that a single neuron (i) takes some set of inputs; (ii) generates a corresponding scalar output; and (iii) has a set of associated parameters that can be updated to optimize some loss function of interest
- then, once we started thinking about networks with multiple outputs, we leveraged vectorized arithmetic to characterize an entire layer of neurons
- just like individual neurons, layers (i) take a set of inputs, (ii) generate corresponding outputs, and (iii) are described by a set of tunable parameters
- when we worked through softmax regression, a single layer was itself the model
- however, even when we subsequently introduced MLPs, we could still think of the model as retaining this same basic structure

- interestingly, for MLPs, both the entire model and its constituent layers share this structure
- the entire model takes in raw inputs (the features), generates outputs (the predictions), and possesses parameters (the combined parameters from all constituent layers)
- likewise, each individual layer ingests inputs (supplied by the previous layer), generates outputs (the inputs to the subsequent layer), and possesses a set of tunable parameters that are updated according to the signal that flows backwards from the subsequent layer

## 2.8 Deep learning computation

- while we might think that neurons, layers, and models give us enough abstractions, it turns out that we often find it convenient to speak about components that are larger than an individual layer, but smaller than the entire model
- for example, the ResNet-152 architecture, which is wildly popular in computer vision, possesses hundreds of layers
- these layers consist of repeating patterns of groups of layers; implementing such a network one layer at a time can grow tedious
- this concern is not just hypothetical – such design patterns are common in practice
- the ResNet architecture, mentioned above, won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection and remains a go-to architecture for many vision tasks
- similar architectures, in which layers are arranged in various repeating patterns, are now ubiquitous in other domains, including natural language processing and speech

## 2.8 Deep learning computation

- to implement these complex networks, we introduce the concept of a neural network *block*
- a block could describe a single layer, a component consisting of multiple layers, or the entire model itself
- one benefit of working with the block abstraction is that they can be combined into larger artifacts, often recursively
- this is illustrated in Figure 15
- by defining code to generate blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks

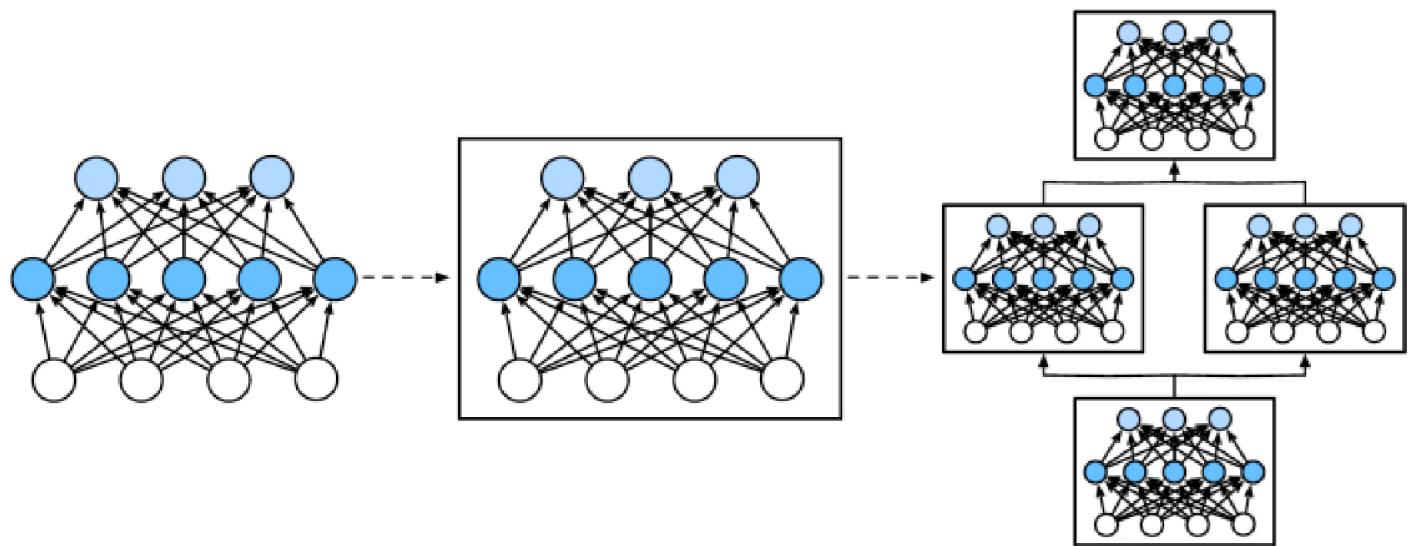


Figure 15: Multiple layers are combined into blocks, forming repeating patterns of larger models.

## 2.8 Deep learning computation

- from a programming point of view, a block is represented by a *class*
- any subclass of it must define a forward propagation function that transforms its input into output and must store any necessary parameters
- note that some blocks do not require any parameters at all
- finally, a block must possess a backpropagation function, for purposes of calculating gradients
- fortunately, due to the auto differentiation feature present in deep learning frameworks, when defining our own block, we only need to worry about parameters and the forward propagation function

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 5

November 1st, 2022

### 3 Convolutional neural networks

- for image data, each example consists of a two-dimensional *grid of pixels*
- depending on whether we are handling black-and-white or color images, each pixel location might be associated with either one or multiple numerical values, respectively
- one way of dealing with this rich structure is to simply discard each image's spatial structure by flattening them into one-dimensional vectors, and feeding them through a fully-connected MLP, which is deeply unsatisfying
- because these networks are invariant to the order of the features, we could get similar results regardless of whether we preserve an order corresponding to the spatial structure of the pixels, or if we permute the columns of our design matrix before fitting the MLP's parameters
- preferably, we would leverage our prior knowledge that nearby pixels are typically related to each other, to build efficient models for learning from image data

### 3 Convolutional neural networks

- in this chapter, we introduce *convolutional neural networks (CNNs)*, a powerful family of neural networks that are designed for precisely this purpose
- CNN-based architectures are now ubiquitous in the field of computer vision, and have become so dominant that hardly anyone today would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without building off of this approach

### 3 Convolutional neural networks

- modern CNNs owe their design to inspirations from biology, group theory, and much experimentation
- in addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient, both because they require fewer parameters than fully-connected architectures, and because convolutions are easy to parallelize across GPU cores
- consequently, CNNs are often applied whenever possible, and increasingly they have emerged as good competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used
- some clever adaptations of CNNs have also brought applications on graph-structured data and in recommender systems

### 3 Convolutional neural networks

- first, we will walk through the basic operations that comprise the backbone of all convolutional networks
- these include the convolutional layers themselves, details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple channels at each layer, and a careful discussion of the structure of modern architectures
- we will discuss the example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning
- then, we will dive into full details of some popular and comparatively recent CNN architectures, whose designs represent most of the techniques commonly used today

### 3.1 From fully-connected layers to convolutions

- to this day, the models that we have discussed so far remain appropriate options when we are dealing with tabular data
- by tabular, we mean that the data consist of rows corresponding to examples and columns corresponding to features
- with tabular data, we might anticipate that the patterns we seek could involve interactions among the features, but we do not assume any structure *a priori* concerning how the features interact
- sometimes, we truly lack knowledge to guide the construction of better architectures; in these cases, an MLP may be the best that we can do
- however, for high-dimensional perceptual data, such structure-less networks may not be suited

### 3.1 From fully-connected layers to convolutions

- for instance, suppose we want to distinguish cats from dogs
- say that we do a thorough job in data collection, collecting an annotated dataset of one-megapixel photographs
- this means that each input to the network has one million dimensions
- according to our discussions of parameterization cost of fully-connected layers, even an aggressive reduction to one thousand hidden dimensions would require a fully-connected layer characterized by  $10^6 \times 10^3 = 10^9$  parameters
- learning the parameters of this network may turn out to be infeasible

### 3.1 From fully-connected layers to convolutions

- we might object to this argument on the basis that one megapixel resolution may not be necessary
- however, while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000 grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters
- moreover, learning a classifier by fitting so many parameters might require collecting an enormous dataset
- and yet, today, both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these intuitions
- that is because images exhibit rich structure that can be exploited by humans and machine learning models alike
- convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images

### 3.1 From fully-connected layers to convolutions

- imagine that we want to detect an object in an image
- it seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image; ideally, our system should exploit this knowledge
- we can draw some inspiration here from the children's game "Where's Waldo" (depicted in Figure 1)
- the game consists of a number of chaotic scenes with many activities

### 3.1 From fully-connected layers to convolutions

- Waldo shows up somewhere in each, typically in some unlikely location; the player's goal is to locate him
- despite his characteristic outfit, this can be surprisingly difficult, due to the large number of distractions
- however, *what Waldo looks like* does not depend upon *where Waldo is located*
- we could sweep the image with a Waldo detector that could assign a score to each patch, indicating the likelihood that the patch contains Waldo
- CNNs systematize this idea of *spatial invariance*, exploiting it to learn useful representations with fewer parameters

### 3.1 From fully-connected layers to convolutions



Figure 1: An image of the "Where's Waldo" game.

### 3.1 From fully-connected layers to convolutions

- we can now make these intuitions more concrete by enumerating a few desiderata to guide our design of a neural network architecture suitable for computer vision:
  - ① In the earliest layers, our network should respond similarly to the same patch, regardless of where it appears in the image. This principle is called *translation invariance*.
  - ② The earliest layers of the network should focus on local regions, without regard for the contents of the image in distant regions. This is the *locality* principle. Eventually, these local representations can be *aggregated* to make predictions at the whole image level.
- let us see how this translates into mathematics; first, we need to discuss the notion of tensors
- just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes
- tensors give us a generic way of describing  $n$ -dimensional arrays with an arbitrary number of axes
- vectors, for example, are first-order tensors, and matrices are second-order tensors
- tensors are denoted with plain capital letters (e.g., X, Y, and Z) and their indexing mechanism (e.g.,  $x_{ijk}$  and  $[X]_{i,j,k}$ ) is similar to that of matrices

### 3.1 From fully-connected layers to convolutions

- to start off, we can consider an MLP with two-dimensional images  $\mathbf{X}$  as inputs and their immediate hidden representations  $\mathbf{H}$  similarly represented as matrices, where both  $\mathbf{X}$  and  $\mathbf{H}$  have the same shape
- we now conceive of not only the inputs, but also the hidden representations as possessing spatial structure
- let  $[\mathbf{X}]_{i,j}$  and  $[\mathbf{H}]_{i,j}$  denote the pixel at location  $(i, j)$  in the input image and hidden representation, respectively
- consequently, to have each of the hidden units receive input from each of the input pixels, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as fourth-order weight tensors  $\mathbf{W}$

### 3.1 From fully-connected layers to convolutions

- suppose that  $\mathbf{U}$  contains biases, we could formally express the fully-connected layer as:

$$\begin{aligned} [\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathbf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathbf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}, \end{aligned}$$

where the switch from  $\mathbf{W}$  to  $\mathbf{V}$  is entirely cosmetic for now, since there is a one-to-one correspondence between coefficients in both fourth-order tensors

- we simply re-index the subscripts  $(k, l)$ , such that  $k = i + a$  and  $l = j + b$ ; in other words, we set  $[\mathbf{V}]_{i,j,a,b} = [\mathbf{W}]_{i,j,k,l}$
- the indices  $a$  and  $b$  run over both positive and negative offsets, covering the entire image
- for any given location  $(i, j)$  in the hidden representation  $[\mathbf{H}]_{i,j}$ , we compute its value by summing over pixels in  $\mathbf{X}$ , centered around  $(i, j)$ , and weighted by  $[\mathbf{V}]_{i,j,a,b}$

### 3.1 From fully-connected layers to convolutions

- now, let us invoke the first principle established above: *translation invariance*
- this implies that a shift in the input  $\mathbf{X}$  should simply lead to a shift in the hidden representation  $\mathbf{H}$
- this is only possible if  $\mathbf{V}$  and  $\mathbf{U}$  do not actually depend on  $(i, j)$ , i.e., we have  $[\mathbf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$  and  $\mathbf{U}$  is a constant, say  $u$
- as a result, we can simplify the definition for  $\mathbf{H}$ :

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a, j+b}.$$

- this is a *convolution*: we are effectively weighting pixels at  $(i + a, j + b)$ , in the vicinity of location  $(i, j)$ , with coefficients  $[\mathbf{V}]_{a,b}$  to obtain the value  $[\mathbf{H}]_{i,j}$
- note that  $[\mathbf{V}]_{a,b}$  needs many fewer coefficients than  $[\mathbf{V}]_{i,j,a,b}$ , since it no longer depends on the location within the image, which is a significant progress

### 3.1 From fully-connected layers to convolutions

- now, let us invoke the second principle: *locality*
- as motivated above, we believe that we should not have to look very far away from location  $(i, j)$  in order to gather relevant information to assess what is going on at  $[\mathbf{H}]_{i,j}$
- this means that, outside some range  $|a| > \Delta$  or  $|b| > \Delta$ , we should set  $[\mathbf{V}]_{a,b} = 0$
- equivalently, we can rewrite  $[\mathbf{H}]_{i,j}$  as:

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a, j+b}. \quad (1)$$

### 3.1 From fully-connected layers to convolutions

- note that (1), in a nutshell, is a *convolutional layer*
- *convolutional neural networks (CNNs)* are a special family of neural networks that contain convolutional layers
- in the deep learning research community,  $\mathbf{V}$  is referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights*, which are often learnable parameters
- when the local region is small, the difference as compared with a fully-connected network can be dramatic
- while, previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations
- the price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation

### 3.1 From fully-connected layers to convolutions

- before going further, we should briefly review why the above operation is called a *convolution*
- in mathematics, the convolution between two functions, say  $f, g : \mathbb{R}^d \rightarrow \mathbb{R}$  is defined as:

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}.$$

- that is, we measure the *overlap* between  $f$  and  $g$  when one function is “flipped” and shifted by  $\mathbf{x}$
- whenever we have discrete objects, the integral turns into a sum
- for instance, for vectors from the set of square summable infinite dimensional vectors with index running over  $\mathbb{Z}$ , we obtain the following definition:

$$(f * g)(i) = \sum_a f(a)g(i - a).$$

### 3.1 From fully-connected layers to convolutions

- for two-dimensional tensors, we have a corresponding sum with indices  $(a, b)$  for  $f$  and  $(i - a, j - b)$  for  $g$ , respectively:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b). \quad (2)$$

- this looks similar to (1), with one major difference
- rather than using  $(i + a, j + b)$ , we are using the difference instead
- note, though, that this distinction is mostly cosmetic, since we can always match the notation between (1) and (2)
- our original definition in (1) more properly describes a *cross-correlation*; we will come back to this in the following section

### 3.1 From fully-connected layers to convolutions

- returning to our Waldo detector, let us see what this looks like
- the convolutional layer picks windows of a given size and weighs intensities according to the filter  $V$ , as demonstrated in Figure 2
- we might aim to learn a model so that wherever the “Waldoness” is highest, we should find a peak in the hidden layer representations

### 3.1 From fully-connected layers to convolutions



Figure 2: Detecting Waldo.

### 3.1 From fully-connected layers to convolutions

- there is just one problem with this approach; so far, we ignored that images consist of 3 channels: red, green, and blue
- in reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel, e.g., with shape  $1024 \times 1024 \times 3$  pixels
- while the first two of these axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location
- we thus index  $X$  as  $[X]_{i,j,k}$
- the convolutional filter has to adapt accordingly; instead of  $[V]_{a,b}$ , we now have  $[V]_{a,b,c}$

### 3.1 From fully-connected layers to convolutions

- moreover, just as our input consists of a third-order tensor, it turns out to be a good idea to similarly formulate our hidden representations as third-order tensors  $H$
- in other words, rather than just having a single hidden representation corresponding to each spatial location, we want an *entire vector of hidden representations* corresponding to each spatial location
- we could think of the hidden representations as comprising a number of two-dimensional grids, stacked on top of each other
- as in the inputs, these are sometimes called *channels*
- they are also sometimes called *feature maps*, as each provides a spatialized set of learned features to the subsequent layer
- intuitively, we might imagine that, at lower layers that are closer to inputs, some channels could become specialized to recognize edges, while others could recognize textures

### 3.1 From fully-connected layers to convolutions

- to support multiple channels in both inputs ( $X$ ) and hidden representations ( $H$ ), we can add a fourth coordinate to  $V$ :  $[V]_{a,b,c,d}$
- correspondingly, the bias  $u$  becomes a vector  $u$
- putting everything together, we have:

$$[H]_{i,j,d} = [u]_d + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [V]_{a,b,c,d} [X]_{i+a,j+b,c}, \quad (3)$$

where  $d$  indexes the *output channels* in the hidden representations  $H$

- the subsequent convolutional layer will go on to take a third-order tensor,  $H$ , as the input
- being more general, (3) is the definition of a *convolutional layer* for *multiple channels*, where  $V$  is a kernel or filter of the layer

### 3.1 From fully-connected layers to convolutions

- there are still many operations that we need to address
- for instance, we need to figure out how to combine all the hidden representations to a single output, e.g., whether there is a Waldo *anywhere* in the image
- we also need to decide how to compute things efficiently, how to combine multiple layers, appropriate activation functions, and how to make reasonable design choices to yield networks that are effective in practice
- we turn to these issues in the next sections

## 3.2 Convolutions for images

- now that we understand how convolutional layers work in theory, we are ready to see how they work in practice
- building on our motivation of convolutional neural networks as efficient architectures for exploring structure in image data, we stick with images as our running example
- recall that, strictly speaking, convolutional layers are a misnomer, since the operations they express are more accurately described as cross-correlations
- based on our descriptions of convolutional layers, in such a layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation

## 3.2 Convolutions for images

- let us ignore channels for now, and see how this works with two-dimensional data and hidden representations
- in Figure 3, the input is a two-dimensional tensor with a height of 3 and width of 3
- we mark the shape of the tensor as  $3 \times 3$  or  $(3, 3)$
- the height and width of the kernel are both 2
- the shape of the *kernel window* (or *convolution window*) is given by the height and width of the kernel (here, it is  $2 \times 2$ )

## 3.2 Convolutions for images

Input	Kernel	Output													
<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td><td>5</td></tr><tr><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	6	7	8	$\ast$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table>	0	1	2	3
0	1	2													
3	4	5													
6	7	8													
0	1														
2	3														
	=	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>19</td><td>25</td></tr><tr><td>37</td><td>43</td></tr></table>	19	25	37	43									
19	25														
37	43														

**Figure 3:** Two-dimensional cross-correlation operation. The shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation:  
 $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$ .

## 3.2 Convolutions for images

- in the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the upper-left corner of the input tensor, and slide it across the input tensor, both from left to right and top to bottom
- when the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied element-wise, and the resulting tensor is summed up, yielding a single scalar value
- this result gives the value of the output tensor at the corresponding location
- here, the output tensor has a height of 2 and width of 2, and the four elements are derived from the two-dimensional cross-correlation operation:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$

$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$

$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

## 3.2 Convolutions for images

- note that, along each axis, the output size is slightly smaller than the input size
- because the kernel has width and height greater than one, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size  $n_h \times n_w$  minus the size of the convolution kernel  $k_h \times k_w$  via:

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

- this is the case, since we need enough space to “shift” the convolution kernel across the image
- later, we will see how to keep the size unchanged by padding the image with zeros around its boundary, so that there is enough space to shift the kernel

## 3.2 Convolutions for images

- a convolutional layer cross-correlates the input and kernel, and adds a scalar bias to produce an output
- the two parameters of a convolutional layer are the *kernel* and the scalar *bias*
- when training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer
- in  $h \times w$  convolution or a  $h \times w$  convolution kernel, the height and width of the convolution kernel are  $h$  and  $w$ , respectively
- we also refer to a convolutional layer with a  $h \times w$  convolution kernel simply as a  $h \times w$  convolutional layer

## 3.2 Convolutions for images

- let us take a moment to discuss a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change
- in this case, the kernel can be constructed as  $[1, -1]$ , i.e., it has a height of 1 and a width of 2
- designing an edge detector by finite differences  $[1, -1]$  is good, if we know this is precisely what we are looking for
- however, as we look at larger kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing, manually
- thus, we need to *learn* these kernels

## 3.2 Convolutions for images

- recall our observation of the correspondence between the cross-correlation and convolution operations
- here, let us continue to consider two-dimensional convolutional layers
- what if such layers perform strict convolution operations as defined in (2), instead of cross-correlations?
- in order to obtain the output of the strict *convolution* operation, we only need to flip the two-dimensional kernel tensor both horizontally and vertically, and then perform the *cross-correlation* operation with the input tensor
- it is noteworthy that, since kernels are learned from data in deep learning, the outputs of convolutional layers remain unaffected, no matter if such layers perform either the strict convolution operations, or the cross-correlation operations

## 3.2 Convolutions for images

- to illustrate this, suppose that a convolutional layer performs *cross-correlation* and learns the kernel in Figure 3, which is denoted as the matrix  $\mathbf{K}$  here
- assuming that other conditions remain unchanged, when this layer performs strict *convolution* instead, the learned kernel  $\mathbf{K}'$  will be the same as  $\mathbf{K}$  after  $\mathbf{K}'$  is flipped both horizontally and vertically
- that is to say, when the convolutional layer performs strict *convolution* for the input in Figure 3 and  $\mathbf{K}'$ , the same output in Figure 3 (cross-correlation of the input and  $\mathbf{K}$ ) will be obtained
- in keeping with standard terminology with deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different
- besides, we use the term *element* to refer to an entry (or component) of any tensor representing a layer representation or a convolution kernel

## 3.2 Convolutions for images

- as described before, the convolutional layer output in Figure 3 is sometimes called a *feature map*, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer
- in CNNs, for any element  $x$  of some layer, its *receptive field* refers to all the elements (from all the previous layers) that may affect the calculation of  $x$  during the forward propagation
- note that the receptive field may be larger than the actual size of the input

## 3.2 Convolutions for images

- let us continue to use Figure 3 to explain the receptive field
- given the  $2 \times 2$  convolution kernel, the *receptive field* of the shaded output element (of value 19) is the four elements in the shaded portion of the input
- now, let us denote the  $2 \times 2$  output as  $\mathbf{Y}$  and consider a deeper CNN with an additional  $2 \times 2$  convolutional layer that takes  $\mathbf{Y}$  as its input, outputting a single element  $z$
- in this case, the receptive field of  $z$  on  $\mathbf{Y}$  includes all the four elements of  $\mathbf{Y}$ , while the receptive field on the input includes all the nine input elements
- thus, when any element in a feature map needs a larger receptive field to detect input features over a broader area, we can build a deeper network

### 3.3 Padding and stride

- in the previous example of Figure 3, our input had both a height and width of 3 and our convolution kernel had both a height and width of 2, yielding an output representation with dimension  $2 \times 2$
- as we generalized in Section 3.2, assuming that the input shape is  $n_h \times n_w$  and the convolution kernel shape is  $k_h \times k_w$ , then the output shape will be  $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel

### 3.3 Padding and stride

- in several cases, we incorporate techniques, including *padding* and *strided convolutions*, that affect the size of the output
- as motivation, note that, since kernels generally have width and height greater than 1, after applying many successive convolutions, we tend to wind up with outputs that are considerably smaller than our input
- if we start with a  $240 \times 240$  pixel image, 10 layers of  $5 \times 5$  convolutions reduce the image to  $200 \times 200$  pixels, slicing off 30% of the image, and with it ignoring any interesting information on the boundaries of the original image
- *padding* is the most popular tool for handling this issue
- in other cases, we may want to reduce the dimensionality drastically, e.g., if we find the original input resolution to be too big
- *strided convolutions* are a popular technique that can help in these instances

### 3.3 Padding and stride

- as described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image
- since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers
- one straightforward solution to this problem is to add extra pixels around the boundary of our input image, thus increasing the effective size of the image
- typically, we set the values of the extra pixels to zero
- in Figure 4, we pad a  $3 \times 3$  input, increasing its size to  $5 \times 5$ ; the corresponding output then increases to a  $4 \times 4$  matrix
- the shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation:  $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$

### 3.3 Padding and stride

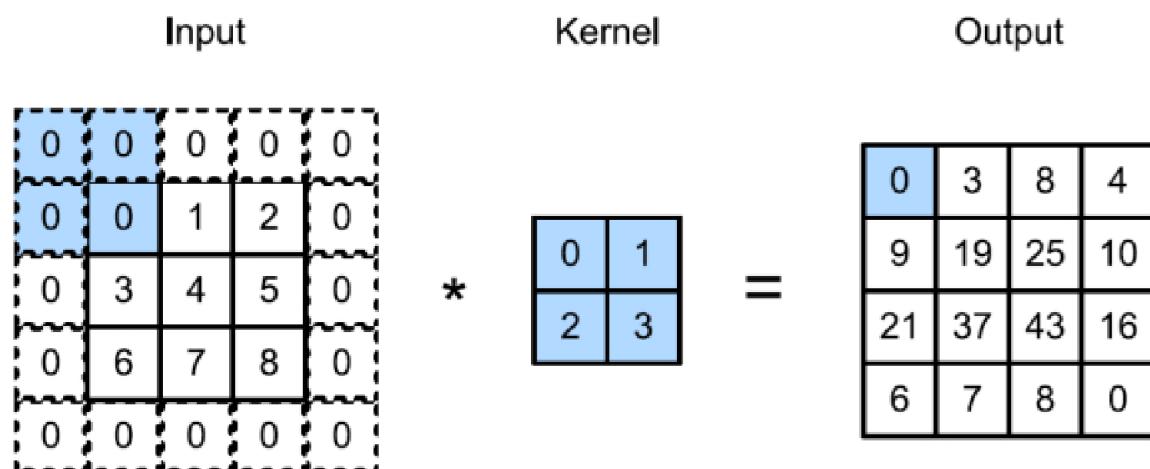


Figure 4: Two-dimensional cross-correlation with padding.

### 3.3 Padding and stride

- in general, if we add a total of  $p_h$  rows of padding (roughly half on top and half on bottom) and a total of  $p_w$  columns of padding (roughly half on the left and half on the right), the output shape will be:
$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1).$$
- this means that the height and width of the output will increase by  $p_h$  and  $p_w$ , respectively
- in many cases, we will want to set  $p_h = k_h - 1$  and  $p_w = k_w - 1$  to give the input and output the same height and width
- this will make it easier to predict the output shape of each layer, when constructing the network
- assuming that  $k_h$  is odd here, we will pad  $p_h/2$  rows on both sides of the height
- if  $k_h$  is even, one possibility is to pad  $\lceil p_h/2 \rceil$  rows on the top of the input and  $\lfloor p_h/2 \rfloor$  rows on the bottom
- we will proceed in the same way for the width

### 3.3 Padding and stride

- CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7
- choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right
- moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clear benefit
- for any two-dimensional tensor  $\mathbf{X}$ , when the kernel's size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output  $[\mathbf{Y}]_{i,j}$  is calculated by cross-correlation of the input and convolution kernel with the window centered on  $[\mathbf{X}]_{i,j}$

### 3.3 Padding and stride

- when computing the cross-correlation, we start with the convolution window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right
- in previous examples, we default to sliding one element at a time
- however, sometimes, either for computational efficiency, or because we wish to downsample, we move our window more than one element at a time, skipping the intermediate locations
- we refer to the number of rows and columns traversed per slide as the *stride*
- so far, we have used strides of 1, both for height and width; sometimes, we may want to use a larger stride

### 3.3 Padding and stride

- Figure 5 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally
- the shaded portions are the output elements, as well as the input and kernel tensor elements used for the output computation:  $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ ,  $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$
- we can see that, when the second element of the first column is outputted, the convolution window slides down three rows
- the convolution window slides two columns to the right when the second element of the first row is outputted
- when the convolution window continues to slide two columns to the right on the input, there is no output, because the input element cannot fill the window (unless we add another column of padding)

### 3.3 Padding and stride

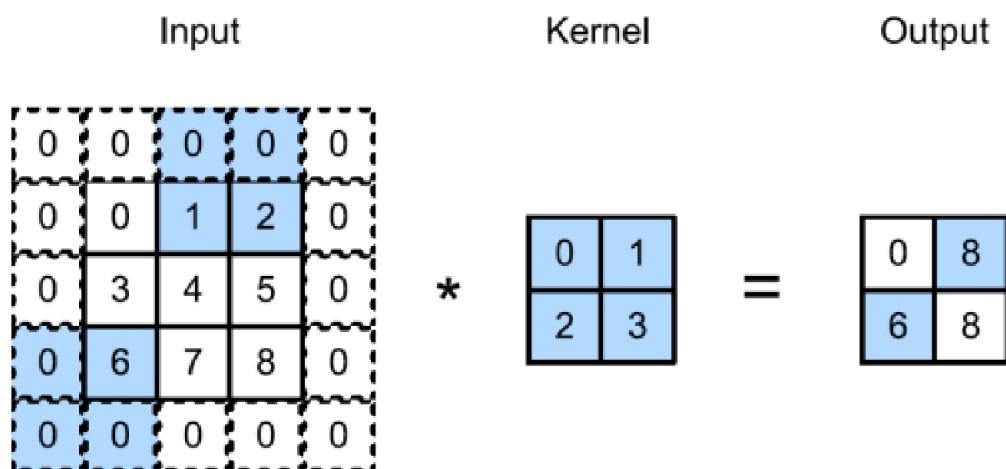


Figure 5: Cross-correlation with strides of 3 and 2 for height and width, respectively.

### 3.3 Padding and stride

- in general, when the stride for the height is  $s_h$  and the stride for the width is  $s_w$ , the output shape is:

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$

- if we set  $p_h = k_h - 1$  and  $p_w = k_w - 1$ , then the output shape will be simplified to  $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$
- going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be  $(n_h/s_h) \times (n_w/s_w)$

### 3.3 Padding and stride

- for the sake of brevity, when the padding number on both sides of the input height and width are  $p_h$  and  $p_w$  respectively, we call the padding  $(p_h, p_w)$
- specifically, when  $p_h = p_w = p$ , the padding is  $p$
- when the strides on the height and width are  $s_h$  and  $s_w$ , respectively, we call the stride  $(s_h, s_w)$
- specifically, when  $s_h = s_w = s$ , the stride is  $s$
- by default, the padding is 0 and the stride is 1
- in practice, we rarely use inhomogeneous strides or padding, i.e., we usually have  $p_h = p_w$  and  $s_h = s_w$

## 3.4 Multiple input and multiple output channels

- while we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green, and blue) and convolutional layers for multiple channels in Section 3.1, until now, we simplified all of our numerical examples by working with just a single input and a single output channel
- this has allowed us to think of our inputs, convolution kernels, and outputs each as two-dimensional tensors
- when we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors; for example, each RGB input image has shape  $3 \times h \times w$
- we refer to this axis, with a size of 3, as the *channel dimension*
- in this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels

## 3.4 Multiple input and multiple output channels

- when the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data
- assuming that the number of channels for the input data is  $c_{in}$ , the number of input channels of the convolution kernel also needs to be  $c_{in}$
- if our convolution kernel's window shape is  $k_h \times k_w$ , then, when  $c_{in} = 1$ , we can think of our convolution kernel as just a two-dimensional tensor of shape  $k_h \times k_w$
- however, when  $c_{in} > 1$ , we need a kernel that contains a tensor of shape  $k_h \times k_w$  for every input channel
- concatenating these  $c_{in}$  tensors together yields a convolution kernel of shape  $c_{in} \times k_h \times k_w$

### 3.4 Multiple input and multiple output channels

- since the input and convolution kernel each have  $c_{in}$  channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the  $c_{in}$  results together (summing over the channels) to yield a two-dimensional tensor
- this is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel
- in Figure 6, we demonstrate an example of a two-dimensional cross-correlation with two input channels
- the shaded portions are the first output element, as well as the input and kernel tensor elements used for the output computation:  
$$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56.$$

### 3.4 Multiple input and multiple output channels

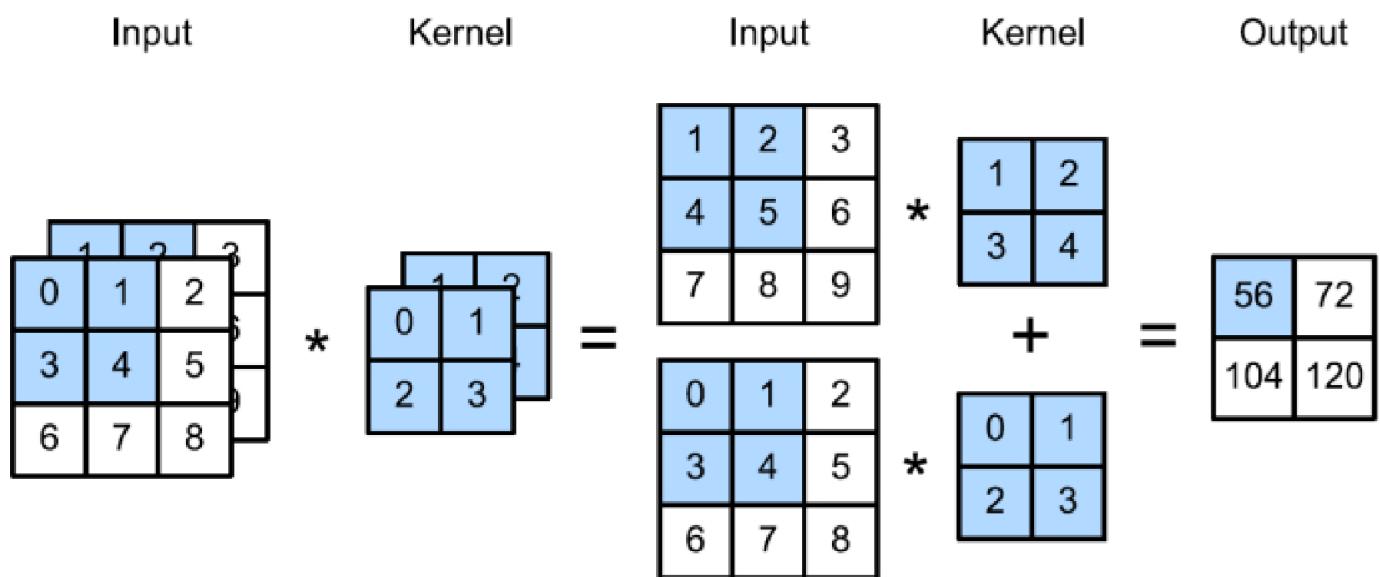


Figure 6: Cross-correlation computation with 2 input channels.

## 3.4 Multiple input and multiple output channels

- regardless of the number of input channels, so far we always ended up with one output channel
- however, as we discussed in Section 3.1, it turns out to be essential to have multiple channels at each layer
- in the most popular neural network architectures, we actually increase the channel dimension as we go higher up in the neural network, typically downsampling to trade off spatial resolution for greater *channel depth*
- intuitively, we could think of each channel as responding to some different set of features
- reality is a bit more complicated than the most naive interpretations of this intuition, since representations are not learned independently, but are rather optimized to be jointly useful
- so it may not be that a single channel learns an edge detector, but rather that some direction in channel space corresponds to detecting edges

### 3.4 Multiple input and multiple output channels

- denote by  $c_{in}$  and  $c_{out}$  the number of input and output channels, respectively, and let  $k_h$  and  $k_w$  be the height and width of the kernel
- to get an output with multiple channels, we can create a kernel tensor of shape  $c_{in} \times k_h \times k_w$  for *every* output channel
- we concatenate them on the output channel dimension, so that the shape of the convolution kernel is  $c_{out} \times c_{in} \times k_h \times k_w$
- in cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel, and takes input from all channels in the input tensor

### 3.4 Multiple input and multiple output channels

- at first, a  $1 \times 1$  convolution, i.e.,  $k_h = k_w = 1$ , does not seem to make much sense
- after all, a convolution correlates adjacent pixels; a  $1 \times 1$  convolution obviously does not
- nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks
- because the minimum window is used, the  $1 \times 1$  convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions
- the only computation of the  $1 \times 1$  convolution occurs on the channel dimension

### 3.4 Multiple input and multiple output channels

- Figure 7 shows the cross-correlation computation using the  $1 \times 1$  convolution kernel with 3 input channels and 2 output channels
- note that the inputs and outputs have the same height and width
- each element in the output is derived from a linear combination of elements *at the same position* in the input image
- we could think of the  $1 \times 1$  convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the  $c_{in}$  corresponding input values into  $c_{out}$  output values
- because this is still a convolutional layer, the weights are tied across pixel location
- thus, the  $1 \times 1$  convolutional layer requires  $c_{out} \times c_{in}$  weights (plus the bias)

### 3.4 Multiple input and multiple output channels

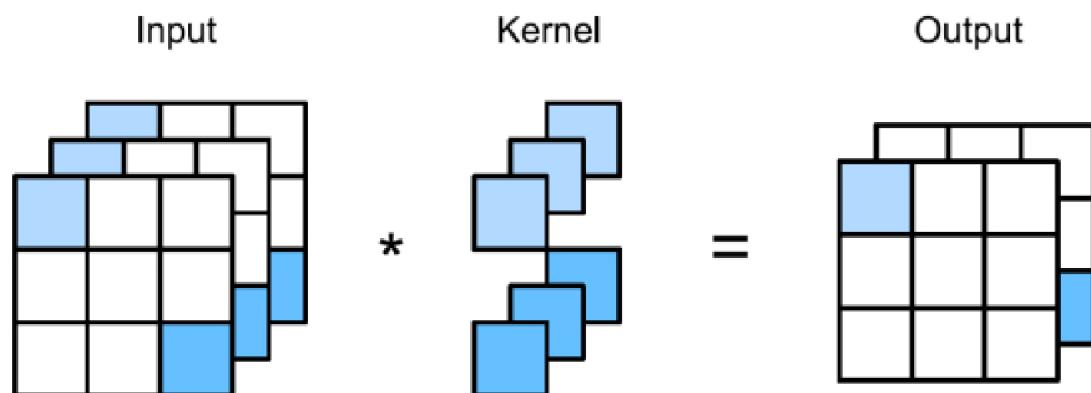


Figure 7: The cross-correlation computation uses the  $1 \times 1$  convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 6

November 8th, 2022

## 3.5 Pooling

- often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations, aggregating information so that the higher up we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive
- often, our ultimate task asks some global question about the image, e.g., *does it contain a cat?*
- so, typically, the units of our final layer should be sensitive to the entire input
- by gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing

## 3.5 Pooling

- moreover, when detecting lower-level features, such as edges, we often want our representations to be somewhat invariant to translation
- for instance, if we take the image  $\mathbf{X}$ , with a sharp delineation between black and white, and shift the whole image by one pixel to the right, i.e.,  $[\mathbf{Z}]_{i,j} = [\mathbf{X}]_{i,j+1}$ , then the output for the new image  $\mathbf{Z}$  might be vastly different
- the edge will have shifted by one pixel; in reality, objects hardly ever occur exactly at the same place
- in fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem)

## 3.5 Pooling

- this section introduces *pooling layers*, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations
- like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*)
- however, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*)

## 3.5 Pooling

- instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window
- these operations are called *maximum pooling* (*max-pooling*, for short) and *average pooling*, respectively
- in both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor, and sliding across the input tensor from left to right and top to bottom
- at each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed

### 3.5 Pooling

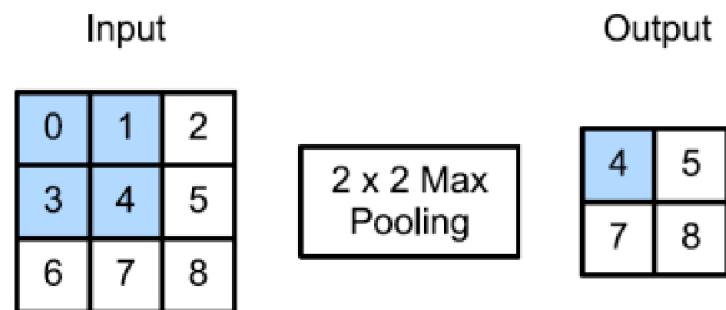


Figure 8: Maximum pooling with a pooling window shape of  $2 \times 2$ . The shaded portions are the first output element, as well as the input tensor elements used for the output computation:  
 $\max(0, 1, 3, 4) = 4$ .

## 3.5 Pooling

- the output tensor in Figure 8 has a height of 2 and a width of 2
- the four elements are derived from the maximum value in each pooling window:

$$\max(0, 1, 3, 4) = 4,$$

$$\max(1, 2, 4, 5) = 5,$$

$$\max(3, 4, 6, 7) = 7,$$

$$\max(4, 5, 7, 8) = 8.$$

- a pooling layer with a pooling window shape of  $p \times q$  is called a  $p \times q$  *pooling layer*
- the pooling operation is called  $p \times q$  *pooling*

## 3.5 Pooling

- let us return to the object edge detection example mentioned at the beginning of this section
- now, we will use the output of the convolutional layer as the input for  $2 \times 2$  maximum pooling
- set the convolutional layer input as  $\mathbf{X}$  and the pooling layer output as  $\mathbf{Y}$
- whether or not the values of  $[\mathbf{X}]_{i,j}$  and  $[\mathbf{X}]_{i,j+1}$  are different, or  $[\mathbf{X}]_{i,j+1}$  and  $[\mathbf{X}]_{i,j+2}$  are different, the pooling layer always outputs  $[\mathbf{Y}]_{i,j} = 1$
- that is to say, using the  $2 \times 2$  maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height or width

## 3.5 Pooling

- as with convolutional layers, pooling layers can also change the output shape
- and, as before, we can alter the operation to achieve a desired output shape by padding the input and adjusting the stride
- when processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels, as in a convolutional layer
- this means that the number of output channels for the pooling layer is the same as the number of input channels

## 3.6 Convolutional neural networks (LeNet)

- we now have all the ingredients required to assemble a fully-functional CNN
- in this section, we will introduce *LeNet*, among the first published CNNs to capture wide attention for its performance on computer vision tasks
- the model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images
- this work represented the culmination of a decade of research developing the technology
- in 1989, LeCun published the first study to successfully train CNNs via backpropagation

## 3.6 Convolutional neural networks (LeNet)

- at the time, LeNet achieved outstanding results, matching the performance of support vector machines, then a dominant approach in supervised learning
- LeNet was eventually adapted to recognize digits for processing deposits in ATM machines; to this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s
- at a high level, LeNet (LeNet-5) consists of two parts:
  - a convolutional encoder consisting of two convolutional layers
  - a dense block consisting of three fully-connected layers; the architecture is summarized in Figure 9

### 3.6 Convolutional neural networks (LeNet)

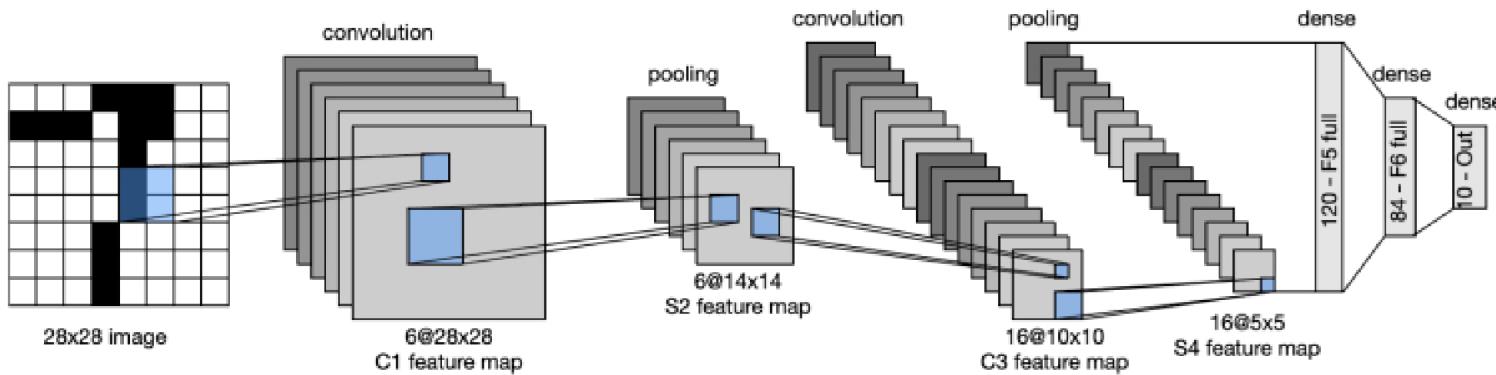


Figure 9: Data flow in LeNet. The input is a handwritten digit and the output is a probability over 10 possible outcomes.

## 3.6 Convolutional neural networks (LeNet)

- the basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation
- note that, while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s
- each convolutional layer uses a  $5 \times 5$  kernel and a sigmoid activation function
- these layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels
- the first convolutional layer has 6 output channels, while the second has 16
- each  $2 \times 2$  pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling
- the convolutional block emits an output with shape given by (batch size, number of channels, height, width)

## 3.6 Convolutional neural networks (LeNet)

- in order to pass output from the convolutional block to the dense block, we must *flatten* each example in the mini-batch
- in other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully-connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the mini-batch and the second to give the flat vector representation of each example
- LeNet's dense block has three fully-connected layers, with 120, 84, and 10 outputs, respectively
- because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes
- by passing a single-channel (black and white)  $28 \times 28$  image through the network, the model is depicted in Figure 10
- we took a small liberty with the original model, removing the Gaussian activation in the final layer; other than that, this network matches the original LeNet-5 architecture

## 3.6 Convolutional neural networks (LeNet)

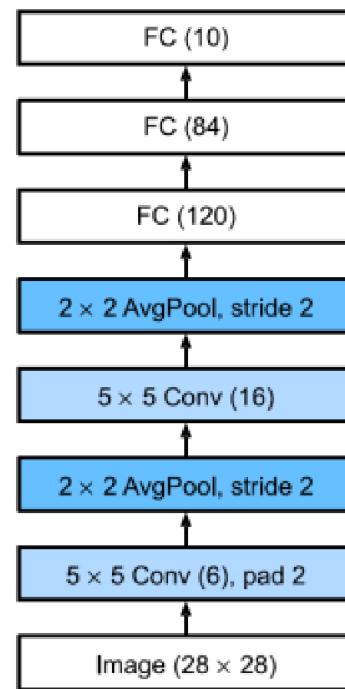


Figure 10: Compressed notation for LeNet-5.

## 3.6 Convolutional neural networks (LeNet)

- note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer)
- the first convolutional layer uses 2 pixels of padding to compensate for the reduction in height and width that would otherwise result from using a  $5 \times 5$  kernel
- in contrast, the second convolutional layer doesn't use padding, and thus the height and width are both reduced by 4 pixels
- as we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer
- however, each pooling layer halves the height and width
- finally, each fully-connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes

### 3.7 Deep convolutional neural networks (AlexNet)

- now that we understand the basics of wiring together CNNs, we will take a tour of modern CNN architectures
- each next section will correspond to a significant CNN architecture that was briefly a dominant architecture and were winners (AlexNet, ResNet) or runners-up (VGG) in the ImageNet competition, which has served as a barometer of progress on supervised learning in computer vision since 2010
- these models include AlexNet, the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge; the VGG network, which makes use of a number of repeating blocks of elements; and residual networks (ResNet), which remain the most popular off-the-shelf architecture in computer vision

### 3.7 Deep convolutional neural networks (AlexNet)

- while the idea of *deep neural networks* is quite simple (stack together a bunch of layers), performance can vary wildly across architectures and hyperparameter choices
- the neural networks described next are the product of intuition, a few mathematical insights, and a whole lot of trial and error
- we present these models in chronological order, partly to convey a sense of the history, so that we can form our own intuitions about where the field is heading, and perhaps develop our own architectures
- for instance, batch normalization and residual connections, described next, have offered two popular ideas for training and designing deep models

### 3.7 Deep convolutional neural networks (AlexNet)

- although CNNs were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field
- although LeNet achieved good results on early small datasets, the performance and feasibility of training CNNs on larger, more realistic datasets had yet to be established
- in fact, for much of the intervening time between the early 1990s and the breakthrough results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines

### 3.7 Deep convolutional neural networks (AlexNet)

- for computer vision, this comparison is perhaps not fair
- that is, although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models
- instead, typical computer vision pipelines consisted of *manually engineering feature extraction* pipelines
- rather than *learn the features*, the features were *crafted*
- most of the progress came from having more clever ideas for features, and the learning algorithm was often very simple

### 3.7 Deep convolutional neural networks (AlexNet)

- although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer CNNs with a large number of parameters
- moreover, datasets were still relatively small
- added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing
- thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:
  - ① Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state-of-the-art).
  - ② Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the discoveries of lucky graduate students.
  - ③ Feed the data through a standard set of feature extractors such as the SIFT (scale-invariant feature transform), the SURF (speeded up robust features), or any number of other hand-tuned pipelines.
  - ④ Dump the resulting representations into a classifier, likely a linear model or kernel method, to train a classifier.

### 3.7 Deep convolutional neural networks (AlexNet)

- another group of researchers, including Yann LeCun, Geoffrey Hinton, Yoshua Bengio, Andrew Ng, and Juergen Schmidhuber had different plans: they believed that features themselves should be learned
- moreover, they believed that, to be reasonably complex, the features should be *hierarchically* composed with multiple jointly learned layers, each with learnable parameters
- in the case of an image, the lowest layers might come to detect edges, colors, and textures
- indeed, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton proposed a new variant of a CNN, *AlexNet*, that achieved excellent performance in the 2012 ImageNet challenge
- AlexNet was named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper

### 3.7 Deep convolutional neural networks (AlexNet)

- interestingly, in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters; Figure 11 is reproduced from the AlexNet paper, and describes lower-level image descriptors

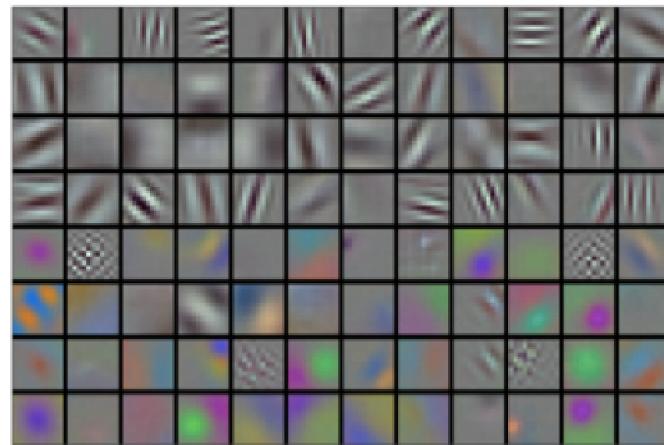


Figure 11: Image filters learned by the first layer of AlexNet.

### 3.7 Deep convolutional neural networks (AlexNet)

- higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and so on
- even higher layers might represent whole objects like people, airplanes, dogs, or frisbees
- ultimately, the final hidden state learns a compact representation of the image that summarizes its contents, such that data belonging to different categories can be easily separated
- while the ultimate breakthrough for many-layered CNNs came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn *hierarchical representations* of visual data for many years
- the ultimate breakthrough in 2012 can be attributed to *two key factors*

### 3.7 Deep convolutional neural networks (AlexNet)

- deep models with many layers require large amounts of *data* in order to enter the regime where they significantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods)
- however, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets
- numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution

### 3.7 Deep convolutional neural networks (AlexNet)

- in 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, each from one of 1000 distinct categories of objects
- the researchers, led by Fei-Fei Li, who introduced this dataset, leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category
- this scale was unprecedented
- the associated competition, named the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered

## 3.7 Deep convolutional neural networks (AlexNet)

- the second factor is *hardware*
- deep learning models are massive consumers of compute cycles
- training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations
- this is one of the main reasons why, in the 1990s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred

### 3.7 Deep convolutional neural networks (AlexNet)

- *graphical processing units* (GPUs) proved to be a game changer in making deep learning feasible
- these chips had long been developed for accelerating graphics processing to benefit computer games
- in particular, they were optimized for high throughput  $4 \times 4$  matrix-vector products, which are needed for many computer graphics tasks
- fortunately, this math is strikingly similar to that required to calculate convolutional layers
- around that time, NVIDIA and ATI had begun optimizing GPUs for general computing operations, going as far as to market them as *general-purpose GPUs* (GPGPU)

### 3.7 Deep convolutional neural networks (AlexNet)

- back to 2012; a major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep CNN that could run on GPU hardware
- they realized that the computational bottlenecks in CNNs, convolutions and matrix multiplications, are all operations that could be parallelized in hardware
- using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions
- the code `cuda-convnet` was good enough that, for several years, it was the industry standard, and powered the first couple years of the deep learning boom

### 3.7 Deep convolutional neural networks (AlexNet)

- AlexNet, which employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin
- this network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision
- the architectures of AlexNet and LeNet are very similar, as Figure 12 illustrates
- note that we provide a slightly streamlined version of AlexNet, removing some of the design tricks that were needed in 2012 to make the model fit on two small GPUs

### 3.7 Deep convolutional neural networks (AlexNet)

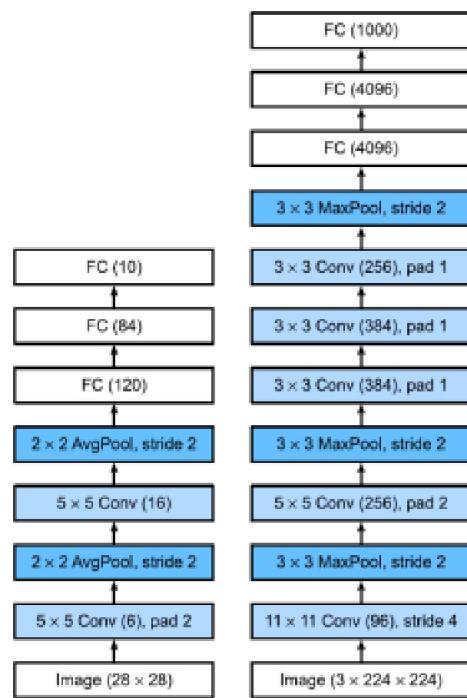


Figure 12: From LeNet (left) to AlexNet (right).

### 3.7 Deep convolutional neural networks (AlexNet)

- the design philosophies of AlexNet and LeNet are very similar, but there are also significant differences
- first, AlexNet is much deeper than the comparatively small LeNet5
- AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer
- second, AlexNet used the ReLU instead of the sigmoid as its activation function

### 3.7 Deep convolutional neural networks (AlexNet)

- in AlexNet's first layer, the convolution window shape is  $11 \times 11$
- since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels
- consequently, a larger convolution window is needed to capture the object
- the convolution window shape in the second layer is reduced to  $5 \times 5$ , followed by  $3 \times 3$
- in addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of  $3 \times 3$  and a stride of 2
- moreover, AlexNet has ten times more convolution channels than LeNet

### 3.7 Deep convolutional neural networks (AlexNet)

- after the last convolutional layer there are two fully-connected layers with 4096 outputs
- these two huge fully-connected layers produce model parameters of nearly 1 GB
- due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model
- fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect)

### 3.7 Deep convolutional neural networks (AlexNet)

- besides, AlexNet changed the sigmoid activation function to a simpler ReLU activation function
- on one hand, the computation of the ReLU activation function is simpler; for example, it does not have the exponentiation operation found in the sigmoid activation function
- on the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods
- this is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that backpropagation cannot continue to update some of the model parameters
- in contrast, the gradient of the ReLU activation function in the positive interval is always 1
- therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained

### 3.7 Deep convolutional neural networks (AlexNet)

- AlexNet controls the model complexity of the fully-connected layer by dropout (Section 2.5), while LeNet only uses weight decay
- to augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes
- this makes the model more robust, and the larger sample size effectively reduces overfitting
- we will discuss data augmentation in greater detail in Chapter 6

### 3.8 Networks using blocks (VGG)

- while AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks
- in the following sections, we will introduce several heuristic concepts commonly used to design deep networks
- progress in this field mirrors that in chip design, where engineers went from placing transistors to logical elements to logic blocks
- similarly, the design of neural network architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers
- the idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, which gave the name of their *VGG* network
- it is easy to implement these repeated structures in code with any modern deep learning framework, by using loops and subroutines

### 3.8 Networks using blocks (VGG)

- the basic building block of classic CNNs is a sequence of the following:
  - a convolutional layer with padding to maintain the resolution
  - a nonlinearity such as a ReLU
  - a pooling layer such as a maximum pooling layer
- one VGG block consists of a sequence of convolutional layers, followed by a maximum pooling layer for spatial downsampling
- in the original VGG paper, the authors employed convolutions with  $3 \times 3$  kernels with padding of 1 (keeping height and width) and  $2 \times 2$  maximum pooling with stride of 2 (halving the resolution after each block)
- like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers, and the second consisting of fully-connected layers; this is depicted in Figure 13

### 3.8 Networks using blocks (VGG)



Figure 13: From AlexNet to VGG, which is designed from building blocks.

## 3.8 Networks using blocks (VGG)

- the convolutional part of the network connects several VGG blocks from Figure 13, in succession
- the fully-connected part of the VGG network is identical to that covered in AlexNet
- the original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each, and the latter three contain two convolutional layers each
- the first block has 64 output channels, and each subsequent block doubles the number of output channels, until that number reaches 512
- since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called *VGG-11*

- training deep neural networks is difficult, and getting them to converge in a reasonable amount of time can be tricky
- in this section, we describe *batch normalization*, a popular and effective technique that consistently accelerates the convergence of deep networks
- together with residual blocks – covered later, in Section 3.10 – batch normalization has made it possible to routinely train networks with over 100 layers
- to motivate batch normalization, let us review a few practical challenges that arise when training machine learning models, and neural networks in particular

### 3.9 Batch normalization

- first, choices regarding data preprocessing often make an enormous difference in the final results
- our first step when working with data is to standardize our input features to each have a mean of zero and variance of one
- intuitively, this standardization plays nicely with our optimizers, because it puts the parameters *a priori* at a similar scale
- second, for a typical MLP or CNN, as we train, the variables (e.g., affine transformation outputs in MLP) in intermediate layers may take values with widely varying magnitudes: both along the layers from the input to the output, across units in the same layer, and over time due to our updates to the model parameters

- the inventors of batch normalization postulated informally that this drift in the distribution of such variables could prevent the convergence of the network
- intuitively, we might conjecture that, if one layer has variable values that are 100 times that of another layer, this might need compensatory adjustments in the learning rates
- third, deeper networks are complex and easily capable of overfitting; this means that regularization becomes more critical

- batch normalization is applied to individual layers (optionally, to all of them) and works as follows: in each training iteration, we first normalize the inputs (of batch normalization) by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current mini-batch
- next, we apply a scale coefficient and a scale offset
- it is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name

### 3.9 Batch normalization

- note that, if we tried to apply batch normalization with mini-batches of size 1, we would not be able to learn anything
- that is because, after subtracting the means, each hidden unit would take value 0
- as we might guess, since we are devoting a whole section to batch normalization, with large enough mini-batches, the approach proves effective and stable
- one takeaway here is that, when applying batch normalization, the choice of batch size may be even more significant than without batch normalization

### 3.9 Batch normalization

- formally, denoting by  $\mathbf{x} \in \mathcal{B}$  an input to batch normalization (BN) that is from a mini-batch  $\mathcal{B}$ , batch normalization transforms  $\mathbf{x}$  according to the following expression:

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}} + \beta. \quad (4)$$

- in (4),  $\hat{\mu}_{\mathcal{B}}$  is the sample mean and  $\hat{\sigma}_{\mathcal{B}}$  is the sample standard deviation of the mini-batch  $\mathcal{B}$
- after applying standardization, the resulting mini-batch has zero mean and unit variance
- because the choice of unit variance (vs. some other number) is an arbitrary choice, we commonly include element-wise *scale parameter*  $\gamma$  and *shift parameter*  $\beta$ , which have the same shape as  $\mathbf{x}$
- note that  $\gamma$  and  $\beta$  are parameters that need to be learned jointly with the other model parameters

## 3.9 Batch normalization

- consequently, the variable magnitudes for intermediate layers cannot diverge during training, because batch normalization actively centers and rescales them back to a given mean and size (via  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$ )
- one piece of practitioner's intuition or wisdom is that batch normalization seems to allow for more aggressive learning rates
- formally, we calculate  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$  in (4) as follows:

$$\begin{aligned}\hat{\mu}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \\ \hat{\sigma}_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\mu}_{\mathcal{B}})^2 + \epsilon.\end{aligned}$$

- note that we add a small constant  $\epsilon > 0$  to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish
- the estimates  $\hat{\mu}_{\mathcal{B}}$  and  $\hat{\sigma}_{\mathcal{B}}$  counteract the scaling issue by using noisy estimates of mean and variance
- we might think that this noisiness should be a problem; as it turns out, this is actually beneficial

### 3.9 Batch normalization

- fixing a trained model, we might think that we would prefer using the entire dataset to estimate the mean and variance
- once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside?
- during training, such exact calculation is infeasible, because the intermediate variables for all data examples change every time we update our model
- however, once the model is trained, we can calculate the means and variances of each layer's variables based on the entire dataset
- indeed, this is standard practice for models employing batch normalization, and thus batch normalization layers function differently in *training mode* (normalizing by mini-batch statistics) and in *prediction mode* (normalizing by dataset statistics)

- we are now ready to take a look at how batch normalization works in practice
- batch normalization implementations for fully-connected layers and convolutional layers are slightly different; we discuss both cases
- recall that one key difference between batch normalization and other layers is that, because batch normalization operates on a full mini-batch at a time, we cannot just ignore the batch dimension, as we did before, when introducing other layers

### 3.9 Batch normalization

- when applying batch normalization to fully-connected layers, the original paper inserts batch normalization after the affine transformation and before the nonlinear activation function (later applications may insert batch normalization right after activation functions)
- denoting the input to the fully-connected layer by  $\mathbf{x}$ , the affine transformation by  $\mathbf{Wx} + \mathbf{b}$  (with the weight parameter  $\mathbf{W}$  and the bias parameter  $\mathbf{b}$ ), and the activation function by  $\phi$ , we can express the computation of a batch-normalization-enabled, fully-connected layer output  $\mathbf{h}$  as follows:

$$\mathbf{h} = \phi(\text{BN}(\mathbf{Wx} + \mathbf{b})).$$

- recall that mean and variance are computed on the *same* mini-batch on which the transformation is applied

### 3.9 Batch normalization

- similarly, with convolutional layers, we can apply batch normalization after the convolution and before the nonlinear activation function
- when the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars
- assume that our mini-batches contain  $m$  examples, and that, for each channel, the output of the convolution has height  $p$  and width  $q$
- for convolutional layers, we carry out each batch normalization over the  $m \cdot p \cdot q$  elements per output channel, simultaneously
- thus, we collect the values over all spatial locations when computing the mean and variance, and consequently apply the same mean and variance within a given channel to normalize the value at each spatial location

### 3.9 Batch normalization

- as we mentioned earlier, batch normalization typically behaves differently in training mode and prediction mode
- first, the noise in the sample mean and the sample variance arising from estimating each on mini-batches are no longer desirable, once we have trained the model
- second, we might not have the luxury of computing per-batch normalization statistics
- for example, we might need to apply our model to make one prediction at a time
- typically, after training, we use the entire dataset to compute stable estimates of the variable statistics, and then fix them at prediction time
- consequently, batch normalization behaves differently during training and at test time; recall that dropout also exhibits this characteristic

## 3.10 Residual networks (ResNet)

- as we design increasingly deeper networks, it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network
- even more important is the ability to design networks where adding layers makes networks strictly more expressive, rather than just different
- this is the question that He et al. considered when working on very deep computer vision models
- at the heart of their proposed *residual network (ResNet)* is the idea that every additional layer should more easily contain the *identity function* as one of its elements
- these considerations are rather profound, but they led to a surprisingly simple solution, the *residual block*
- with it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015
- the design had a profound influence on how to build deep neural networks

## 3.10 Residual networks (ResNet)

- let us focus on a local part of a neural network, as depicted in Figure 14
- denote the input by  $\mathbf{x}$ , and assume that the desired underlying mapping we want to obtain by learning is  $f(\mathbf{x})$ , to be used as the input to the activation function on the top
- on the left of Figure 14, the portion within the dotted-line box must directly learn the mapping  $f(\mathbf{x})$
- on the right, the portion within the dotted-line box needs to learn the *residual mapping*  $f(\mathbf{x}) - \mathbf{x}$ , which is how the residual block derives its name
- if the identity mapping  $f(\mathbf{x}) = \mathbf{x}$  is the desired underlying mapping, the residual mapping is easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully-connected layer and convolutional layer) within the dotted-line box to zero
- the right figure in Figure 14 illustrates the *residual block* of ResNet, where the solid line carrying the layer input  $\mathbf{x}$  to the addition operator is called a *residual connection* (or *shortcut connection*)
- with residual blocks, inputs can forward propagate faster through the residual connections across layers

## 3.10 Residual networks (ResNet)

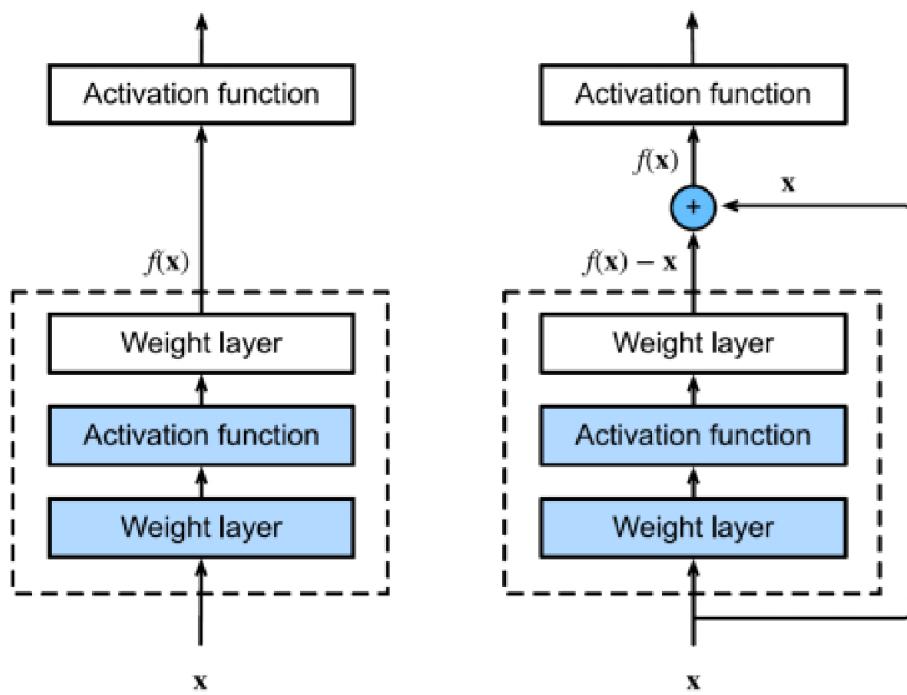


Figure 14: A regular block (left) and a residual block (right).

## 3.10 Residual networks (ResNet)

- ResNet follows VGG's full  $3 \times 3$  convolutional layer design
- the residual block has two  $3 \times 3$  convolutional layers with the same number of output channels
- each convolutional layer is followed by a batch normalization layer and a ReLU activation function
- then, we skip these two convolution operations and add the input directly before the final ReLU activation function
- this kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together
- if we want to change the number of channels or the resolution, we need to introduce an additional  $1 \times 1$  convolutional layer to transform the input into the desired shape for the addition operation
- thus, we have two types of blocks: one where we add the input to the output before applying the ReLU nonlinearity, and one where we adjust channels and resolution by means of a  $1 \times 1$  convolution before adding; Figure 15 illustrates this

### 3.10 Residual networks (ResNet)

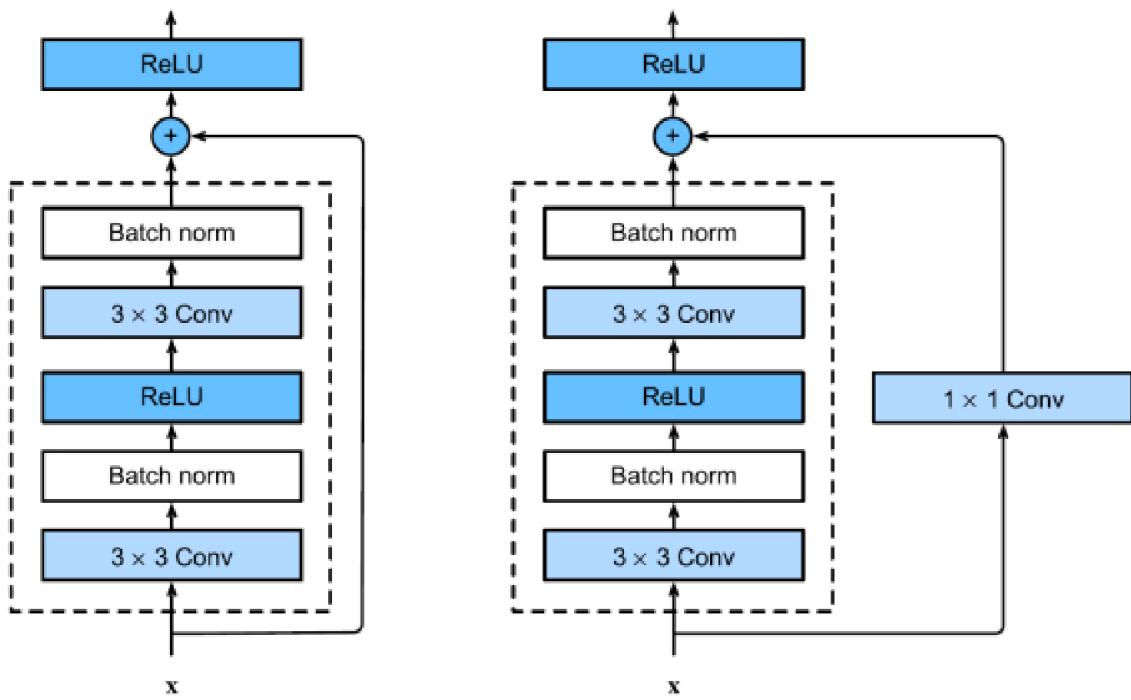


Figure 15: ResNet block with and without  $1 \times 1$  convolution.

## 3.10 Residual networks (ResNet)

- the first two layers of ResNet are a  $7 \times 7$  convolutional layer with 64 output channels and a stride of 2, which is followed by the  $3 \times 3$  maximum pooling layer with a stride of 2
- a batch normalization layer is added after each convolutional layer in ResNet
- ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels
- the number of channels in the first module is the same as the number of input channels
- since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width
- in the first residual block, for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved

## 3.10 Residual networks (ResNet)

- finally, we add a global average pooling layer, followed by the fully-connected layer output
- there are 4 convolutional layers in each module (excluding the  $1 \times 1$  convolutional layer)
- together with the first  $7 \times 7$  convolutional layer and the final fully-connected layer, there are 18 layers in total
- therefore, this model is commonly known as *ResNet-18*
- by configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer *ResNet-152*
- the main architecture of ResNet is simple and easy to modify, which has resulted in the rapid and widespread use of ResNet; Figure 16 depicts the full ResNet-18

## 3.10 Residual networks (ResNet)

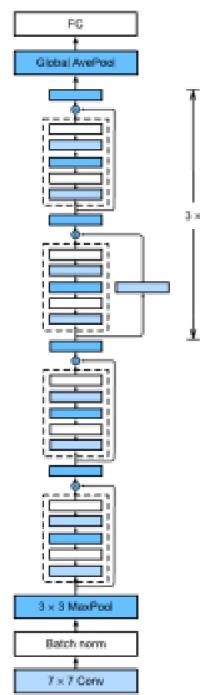


Figure 16: The ResNet-18 architecture.

# Thank you!



# 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

## 4 Recurrent neural networks

- 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, ...
- 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

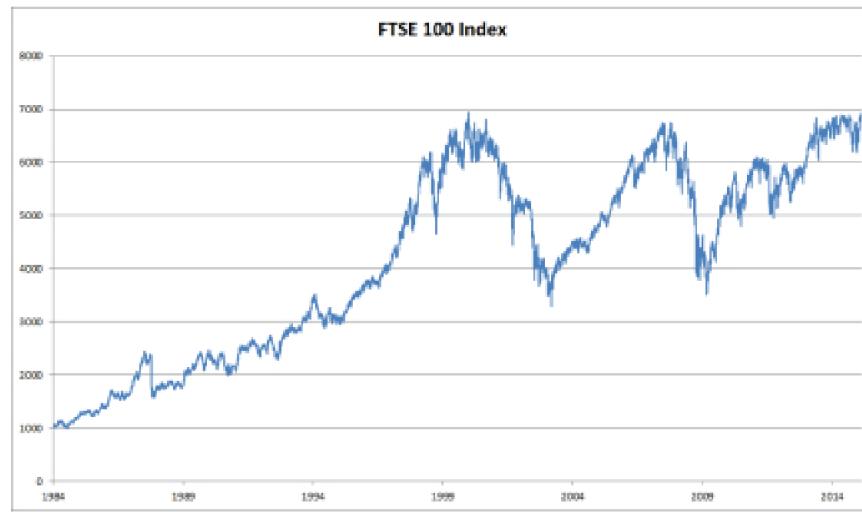


Figure 1: FTSE 100 index over about 30 years.

## 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:

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

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

## 4.1 Sequence models

- there is just one major problem: the number of inputs  $x_{t-1}, \dots, x_1$  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  $P(x_t|x_{t-1}, \dots, x_1)$  efficiently
- in a nutshell, it boils down to two strategies, as follows

## 4.1 Sequence models

- first, assume that the potentially rather long sequence  $x_{t-1}, \dots, x_1$  is not really necessary
- in this case, it might be sufficient to use some timespan of length  $\tau$ , and only use the  $x_{t-\tau}, \dots, x_{t-1}$  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*

## 4.1 Sequence models

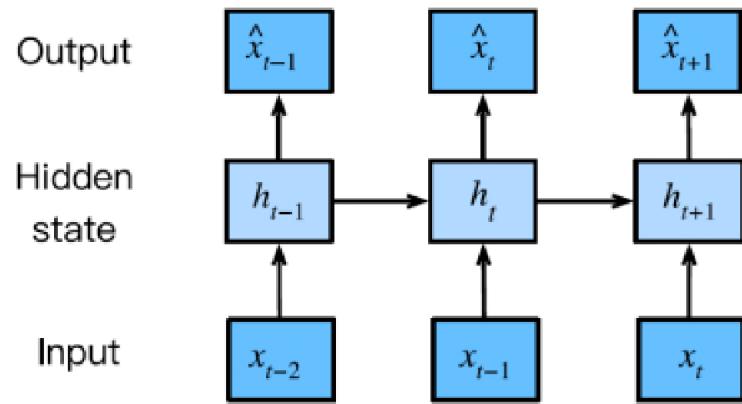


Figure 2: A latent autoregressive model.

## 4.1 Sequence models

- 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:

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

## 4.1 Sequence models

- 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  $P(x_t|x_{t-1}, \dots, x_1)$
- recall the approximation that, in an autoregressive model, we use only  $x_{t-1}, \dots, x_{t-\tau}$  instead of  $x_{t-1}, \dots, x_1$  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:

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

## 4.1 Sequence models

- 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

## 4.1 Sequence models

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

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

- 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

## 4.1 Sequence models

- 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 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 *t*okens (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

## 4.1 Sequence models

- 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  $x_1, x_2, \dots, x_T$
- 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  $P(x_1, x_2, \dots, x_T)$

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

## 4.1 Sequence models

- let us now apply Markov models to language modeling
- a distribution over sequences satisfies the Markov property of first order if  $P(x_{t+1}|x_t, \dots, x_1) = P(x_{t+1}|x_t)$
- 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

## 4.2 Recurrent neural networks

- 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  $P(x_t|x_{t-1}, \dots, x_{t-n+1})$ , it is preferable to use a latent variable model:

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

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  $h_{t-1}$ :

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

- 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

## 4.2 Recurrent neural networks

- 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  $\mathbf{X} \in \mathbb{R}^{n \times d}$  with batch size  $n$  and  $d$  inputs, the hidden layer's output  $\mathbf{H} \in \mathbb{R}^{n \times h}$  is calculated as:

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

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

## 4.2 Recurrent neural networks

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

$$O = HW_{hq} + b_q,$$

where  $O \in \mathbb{R}^{n \times q}$  is the output variable,  $W_{hq} \in \mathbb{R}^{h \times q}$  is the weight parameter, and  $b_q \in \mathbb{R}^{1 \times q}$  is the bias parameter of the output layer

- if it is a classification problem, we can use  $\text{softmax}(O)$  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

## 4.2 Recurrent neural networks

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

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

## 4.2 Recurrent neural networks

- compared with (2), (3) adds one more term  $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ , and thus instantiates (1)
- from the relationship between hidden variables  $\mathbf{H}_t$  and  $\mathbf{H}_{t-1}$  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*

## 4.2 Recurrent neural networks

- 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:

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

- parameters of the RNN include the weights  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ,  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ , and the bias  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$  of the hidden layer, together with the weights  $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ , and the bias  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  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  $\mathbf{X}_t$  at the current time step  $t$  and the hidden state  $\mathbf{H}_{t-1}$  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  $\mathbf{H}_t$  of the current time step  $t$

## 4.2 Recurrent neural networks

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

## 4.2 Recurrent neural networks

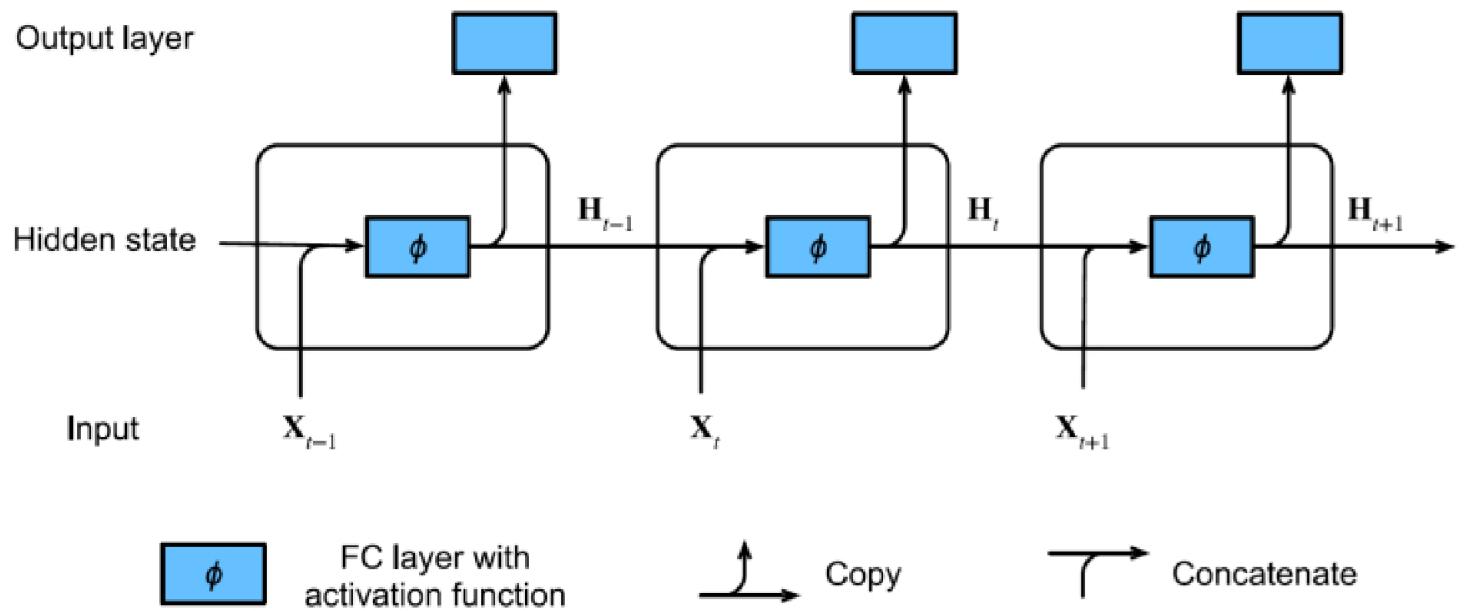


Figure 3: An RNN with a hidden state.

- 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  $\mathbf{x}$ , in the direction of the negative gradient  $\mathbf{g}$ , on a mini-batch
- for example, with  $\eta > 0$  as the learning rate, in one iteration, we update  $\mathbf{x}$  as  $\mathbf{x} - \eta \mathbf{g}$
- let us further assume that the objective function  $f$  is well behaved, say, *Lipschitz continuous* with constant  $L$
- that is to say, for any  $\mathbf{x}$  and  $\mathbf{y}$ , we have:

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

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

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

which means that we will not observe a change by more than  $L\eta \|\mathbf{g}\|$

- 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

## 4.2 Recurrent neural networks

- 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  $\mathbf{g}$  by projecting it back to a ball of a given radius, say  $\theta$ , via:

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

- 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  $\mathbf{g}$
- 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

## 4.2 Recurrent neural networks

- 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

## 4.2 Recurrent neural networks

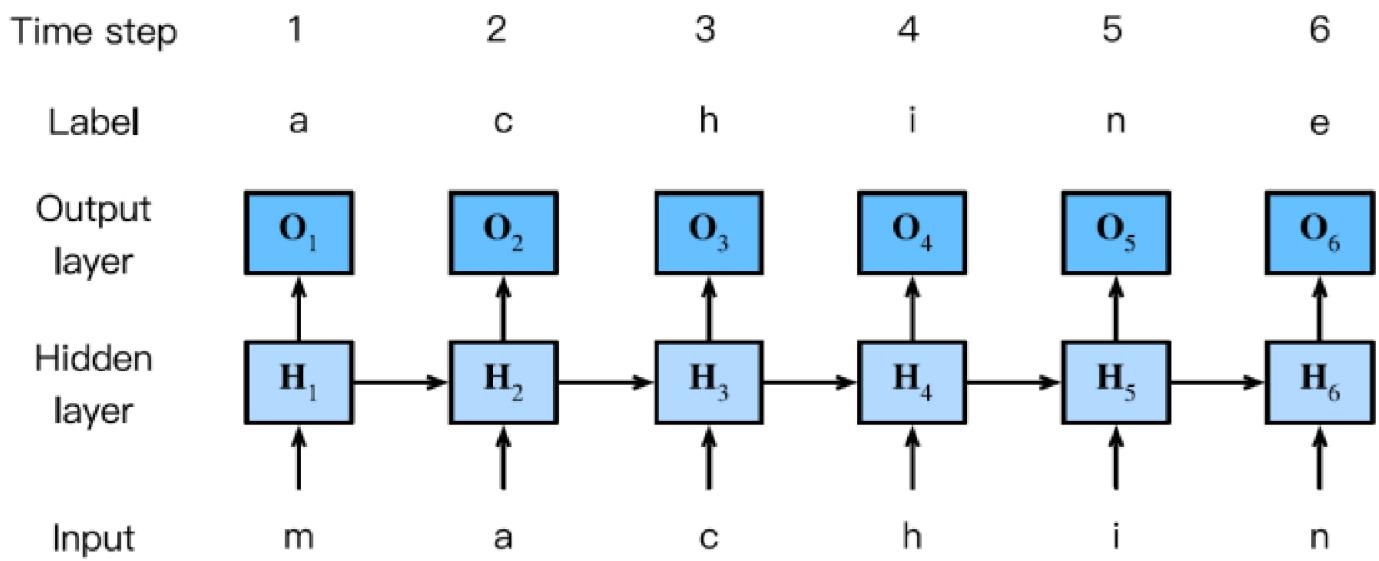


Figure 4: A character-level language model based on the RNN. The input and label sequences are "machin" and "achine", respectively.

## 4.2 Recurrent neural networks

- 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,  $O_3$ , 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

## 4.2 Recurrent neural networks

- in practice, each token is represented by a  $d$ -dimensional vector, and we use a batch size  $n > 1$
- therefore, the input  $\mathbf{X}_t$  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”

## 4.2 Recurrent neural networks

- 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

## 4.2 Recurrent neural networks

- 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

## 4.2 Recurrent neural networks

- 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

## 4.2 Recurrent neural networks

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

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

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):

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

## 4.2 Recurrent neural networks

- 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!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 8

November 29th, 2022

## 4.3 Gated recurrent units (GRU)

- we have introduced the basics of RNNs, which can better handle sequence data
- however, such techniques may not be sufficient for a wide range of sequence learning problems from nowadays
- for instance, a notable issue in practice is the numerical instability of RNNs
- although we have discussed implementation tricks such as gradient clipping, this issue can be alleviated further with more sophisticated designs of sequence models
- specifically, gated RNNs are much more common in practice

## 4.3 Gated recurrent units (GRU)

- we will introduce two such widely-used networks, namely gated recurrent units (GRUs) and long short-term memory (LSTM)
- furthermore, we will expand the RNN architecture with a single hidden layer, which has been discussed so far, and describe deep architectures, with multiple hidden layers
- such expansions are frequently adopted in modern recurrent networks

## 4.3 Gated recurrent units (GRU)

- in Chapter 2, we discussed that long products of matrices can lead to vanishing or exploding gradients
- let us briefly think about what such gradient anomalies mean in practice:
  - We might encounter a situation where an early observation is highly significant for predicting all future observations. Consider the case where the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence. In this case, the influence of the first token is vital. We would like to have some mechanisms for storing vital early information in a *memory cell*. Without such a mechanism, we will have to assign a very large gradient to this observation, since it affects all the subsequent observations.
  - We might encounter situations where some tokens carry no pertinent observation. For instance, when parsing a web page, there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for *skipping* such tokens in the latent state representation.
  - We might encounter situations where there is a logical break between parts of a sequence. For instance, there might be a transition between chapters in a book, or a transition between a bear and a bull market for stocks. In this case, it would be nice to have a means of *resetting* our internal state representation.

## 4.3 Gated recurrent units (GRU)

- a number of methods have been proposed to address this
- one of the earliest is long short-term memory, proposed in 1997, which we will discuss in Section 4.4
- the gated recurrent unit (GRU), proposed in 2014, is a slightly lighter variant that often offers comparable performance and is significantly faster to compute
- due to its simplicity, we will start with the GRU

## 4.3 Gated recurrent units (GRU)

- the key distinction between vanilla RNNs and GRUs is that the latter support gating of the hidden state
- this means that we have dedicated mechanisms for when a hidden state should be *updated*, and also when it should be *reset*
- these mechanisms are learned, and they address the concerns listed above
- for instance, if the first token is of great importance, we will learn not to update the hidden state after the first observation
- likewise, we will learn to skip irrelevant temporary observations
- lastly, we will learn to reset the latent state whenever needed; we discuss this in detail below

## 4.3 Gated recurrent units (GRU)

- the first thing we need to introduce are the *reset gate* and the *update gate*
- we engineer them to be vectors with entries in  $(0, 1)$ , such that we can perform convex combinations
- for instance, a reset gate would allow us to control how much of the previous state we might still want to remember
- likewise, an update gate would allow us to control how much of the new state is just a copy of the old state

## 4.3 Gated recurrent units (GRU)

- we begin by engineering these gates
- Figure 5 illustrates the inputs for both the reset and update gates in a GRU, given the input of the current time step and the hidden state of the previous time step
- the outputs of the two gates are given by two fully-connected layers with a sigmoid activation function

## 4.3 Gated recurrent units (GRU)

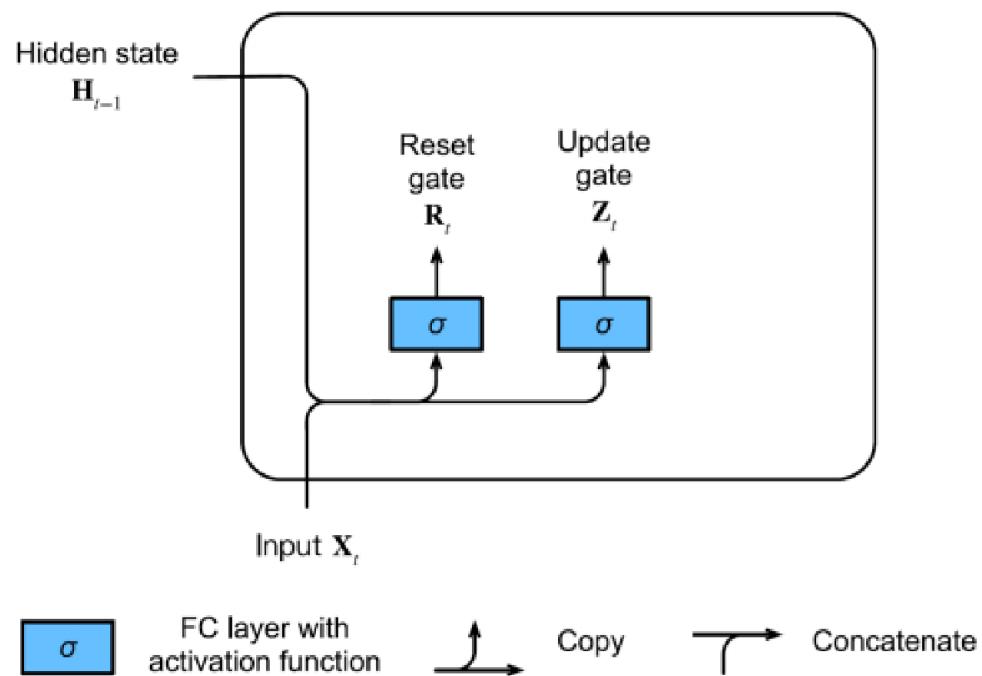


Figure 5: Computing the reset gate and the update gate in a GRU model.

## 4.3 Gated recurrent units (GRU)

- mathematically, for a given time step  $t$ , suppose that the input is a mini-batch  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (number of examples:  $n$ , number of inputs:  $d$ ) and the hidden state of the previous time step is  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$  (number of hidden units:  $h$ )
- then, the *reset gate*  $\mathbf{R}_t \in \mathbb{R}^{n \times h}$  and *update gate*  $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$  are computed as follows:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),\end{aligned}$$

where  $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$  and  $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$  are weight parameters, and  $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$  are biases

- note that broadcasting is triggered during the summation
- we use the sigmoid function  $\sigma$  to transform input values to the interval  $(0, 1)$

## 4.3 Gated recurrent units (GRU)

- next, let us integrate the reset gate  $\mathbf{R}_t$  with the regular latent state updating mechanism in (3)
- it leads to the following *candidate hidden state*  $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  at time step  $t$ :

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h), \quad (5)$$

where  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$  and  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  are weight parameters,  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$  is the bias, and the symbol  $\odot$  is the Hadamard (element-wise) product operator

- here, we use a nonlinearity in the form of tanh to ensure that the values in the candidate hidden state remain in the interval  $(-1, 1)$

## 4.3 Gated recurrent units (GRU)

- the result is a *candidate*, since we still need to incorporate the action of the update gate
- comparing with (3), now, the influence of the previous states can be reduced with the element-wise multiplication of  $R_t$  and  $H_{t-1}$  in (5)
- whenever the entries in the reset gate  $R_t$  are close to 1, we recover a vanilla RNN, such as in (3)
- for all entries of the reset gate  $R_t$  that are close to 0, the candidate hidden state is the result of an MLP with  $X_t$  as the input
- any pre-existing hidden state is thus *reset* to defaults
- Figure 6 illustrates the computational flow after applying the reset gate

## 4.3 Gated recurrent units (GRU)

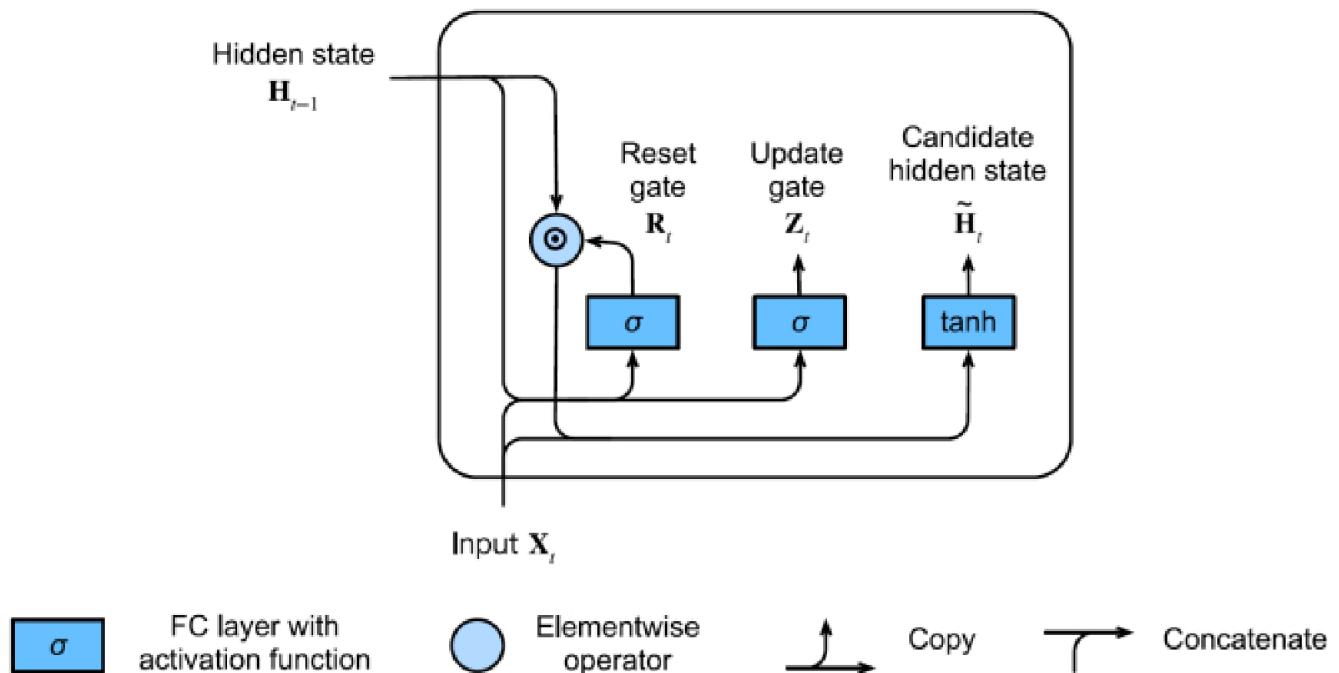


Figure 6: Computing the candidate hidden state in a GRU model.

## 4.3 Gated recurrent units (GRU)

- finally, we need to incorporate the effect of the update gate  $Z_t$
- this determines the extent to which the new hidden state  $H_t \in \mathbb{R}^{n \times h}$  is just the old state  $H_{t-1}$ , and by how much the new candidate state  $\tilde{H}_t$  is used
- the update gate  $Z_t$  can be used for this purpose, simply by taking element-wise convex combinations between both  $H_{t-1}$  and  $\tilde{H}_t$
- this leads to the final update equation for the GRU:

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t.$$

- whenever the update gate  $Z_t$  is close to 1, we simply retain the old state
- in this case, the information from  $X_t$  is essentially ignored, effectively skipping time step  $t$  in the dependency chain

## 4.3 Gated recurrent units (GRU)

- in contrast, whenever  $Z_t$  is close to 0, the new latent state  $H_t$  approaches the candidate latent state  $\tilde{H}_t$
- these designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for sequences with large time step distances
- for instance, if the update gate has been close to 1 for all the time steps of an entire subsequence, the old hidden state at the time step of its beginning will be easily retained and passed to its end, regardless of the length of the subsequence
- Figure 7 illustrates the computational flow after the update gate is in action
- in summary, GRUs have the following two distinguishing features:
  - Reset gates help capture short-term dependencies in sequences.
  - Update gates help capture long-term dependencies in sequences.

## 4.3 Gated recurrent units (GRU)

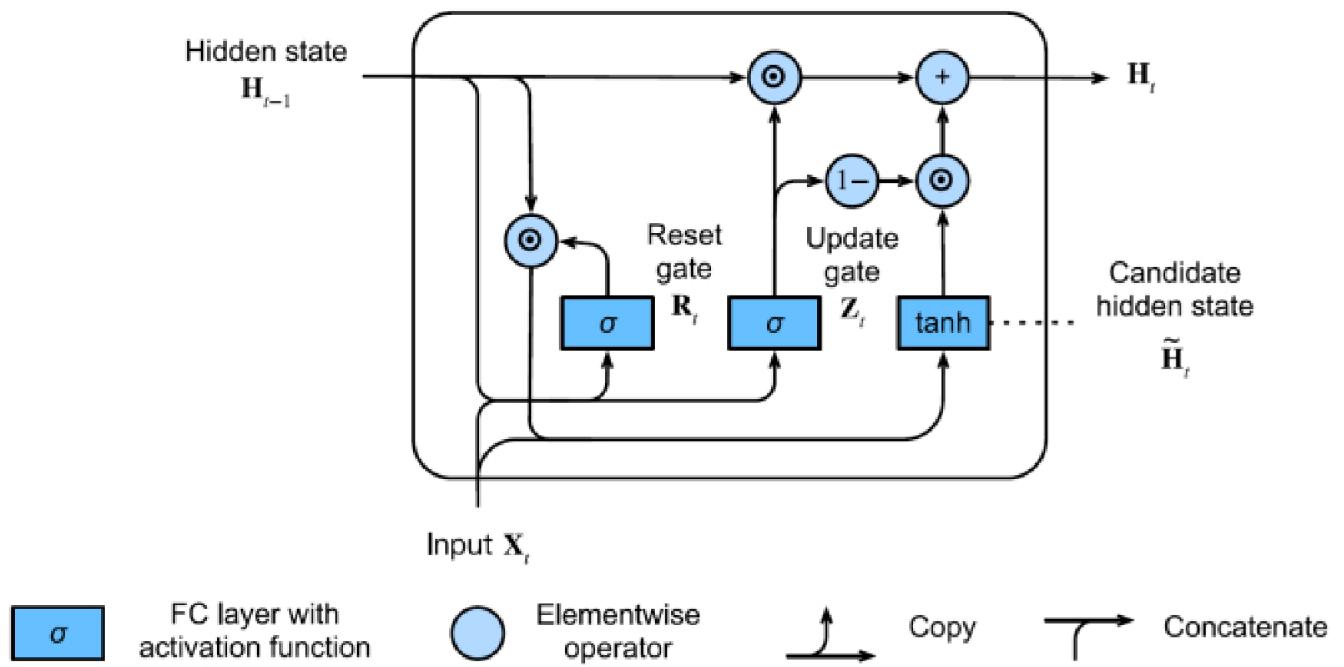


Figure 7: Computing the hidden state in a GRU model.

## 4.4 Long short-term memory (LSTM)

- the challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time
- one of the earliest approaches to address this was the *long short-term memory (LSTM)*
- it shares many of the properties of the GRU
- interestingly, LSTMs have a slightly more complex design than GRUs, but predate GRUs by almost two decades

## 4.4 Long short-term memory (LSTM)

- arguably LSTM's design is inspired by logic gates of a computer
- LSTM introduces a *memory cell* (or *cell* for short) that has the same shape as the hidden state (some papers consider the memory cell as a special type of hidden state), engineered to record additional information
- to control the memory cell, we need a number of gates
- one gate is needed to read out the entries from the cell; we will refer to this as the *output gate*

## 4.4 Long short-term memory (LSTM)

- a second gate is needed to decide when to read data into the cell; we refer to this as the *input gate*
- lastly, we need a mechanism to reset the content of the cell, governed by a *forget gate*
- the motivation for such a design is the same as that of GRUs, namely to be able to decide when to remember and when to ignore inputs in the hidden state, via a dedicated mechanism
- let us see how this works in practice

## 4.4 Long short-term memory (LSTM)

- just like in GRUs, the data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Figure 8
- they are processed by three fully-connected layers with a sigmoid activation function to compute the values of the input, forget, and output gates
- as a result, values of the three gates are in the range of  $(0, 1)$

## 4.4 Long short-term memory (LSTM)

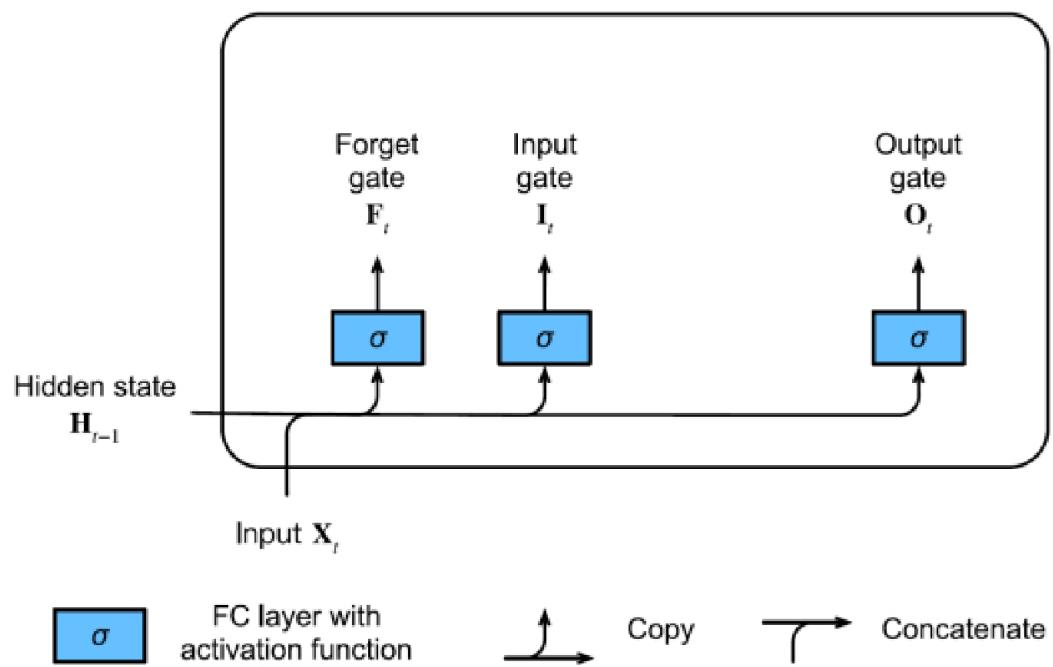


Figure 8: Computing the input gate, the forget gate, and the output gate in an LSTM model.

## 4.4 Long short-term memory (LSTM)

- mathematically, suppose that there are  $h$  hidden units, the batch size is  $n$ , and the number of inputs is  $d$
- thus, the input is  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  and the hidden state of the previous time step is  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$
- correspondingly, the gates at time step  $t$  are defined as follows: the input gate is  $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ , the forget gate is  $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ , and the output gate is  $\mathbf{O}_t \in \mathbb{R}^{n \times h}$
- they are calculated as follows:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}$$

where  $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$  and  $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$  are weight parameters, and  $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$  are bias parameters

## 4.4 Long short-term memory (LSTM)

- next, we design the memory cell
- since we have not specified the action of the various gates yet, we first introduce the *candidate* memory cell  $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$
- its computation is similar to that of the three gates described above, but using a tanh function, with a value range of  $(-1, 1)$  as the activation function
- this leads to the following equation at time step  $t$ :

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

where  $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$  and  $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$  are weight parameters, and  $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$  is a bias parameter

- a quick illustration of the candidate memory cell is shown in Figure 9

## 4.4 Long short-term memory (LSTM)

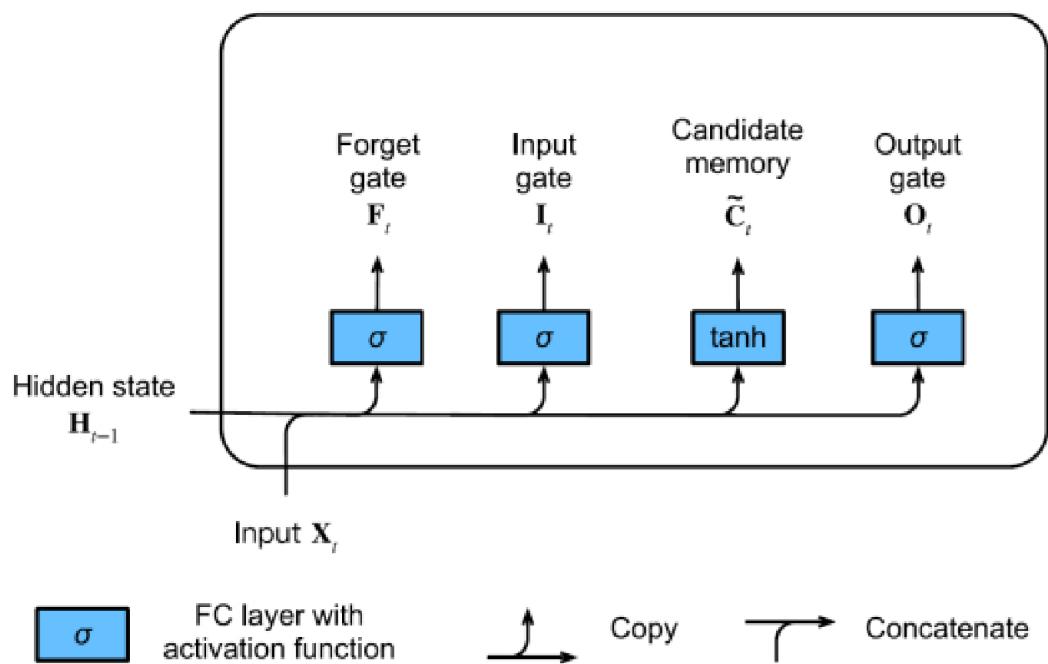


Figure 9: Computing the candidate memory cell in an LSTM model.

## 4.4 Long short-term memory (LSTM)

- in GRUs, we have a mechanism to govern input and forgetting (or skipping)
- similarly, in LSTMs we have two dedicated gates for such purposes: the input gate  $I_t$  governs how much we take new data into account via  $\tilde{C}_t$ , and the forget gate  $F_t$  addresses how much of the old memory cell content  $C_{t-1} \in \mathbb{R}^{n \times h}$  we retain
- using the same element-wise multiplication trick as before, we arrive at the following update equation:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t.$$

- if the forget gate is always approximately 1, and the input gate is always approximately 0, the past memory cells  $C_{t-1}$  will be saved over time and passed to the current time step
- this design is introduced to alleviate the vanishing gradient problem, and to better capture long range dependencies within sequences
- we thus arrive at the flow diagram in Figure 10

## 4.4 Long short-term memory (LSTM)

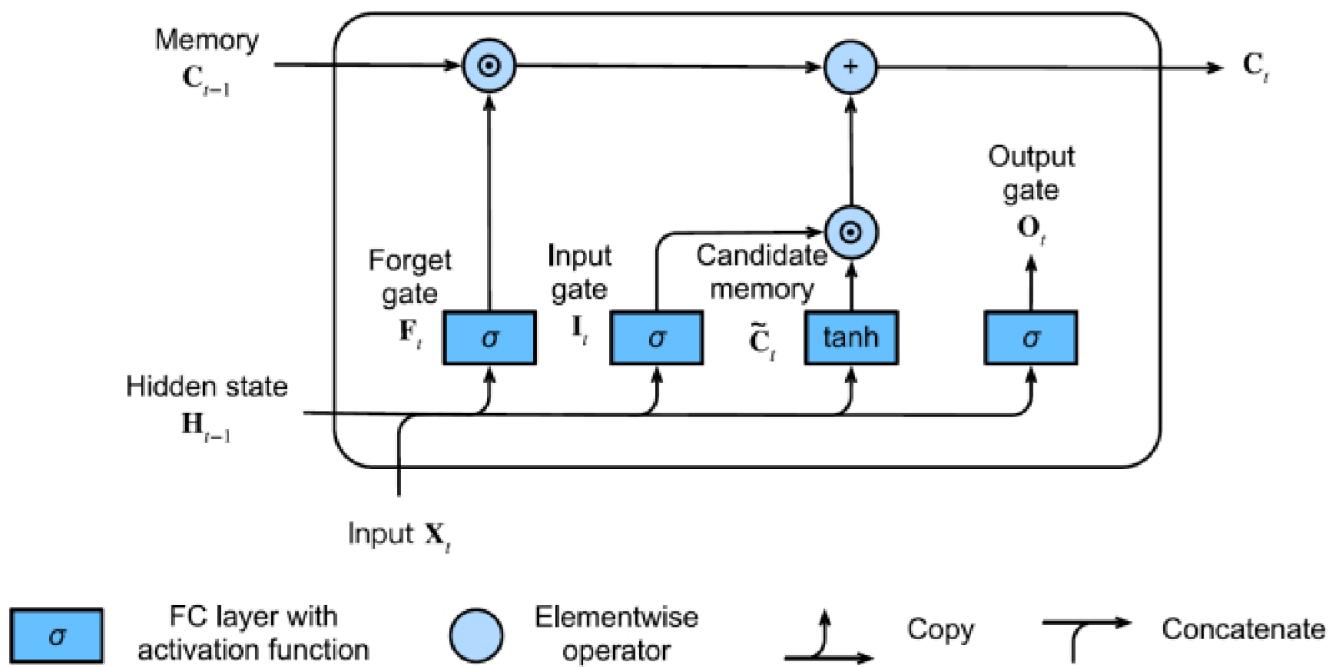


Figure 10: Computing the memory cell in an LSTM model.

## 4.4 Long short-term memory (LSTM)

- lastly, we need to define how to compute the hidden state  $H_t \in \mathbb{R}^{n \times h}$ ; this is where the output gate comes into play
- in LSTM, it is simply a gated version of the tanh of the memory cell
- this ensures that the values of  $H_t$  are always in the interval  $(-1, 1)$ :

$$H_t = O_t \odot \tanh(C_t).$$

- whenever the output gate approximates 1, we effectively pass all memory information through to the predictor, whereas for the output gate close to 0, we retain all the information only within the memory cell, and perform no further processing
- Figure 11 gives a graphical illustration of the data flow

## 4.4 Long short-term memory (LSTM)

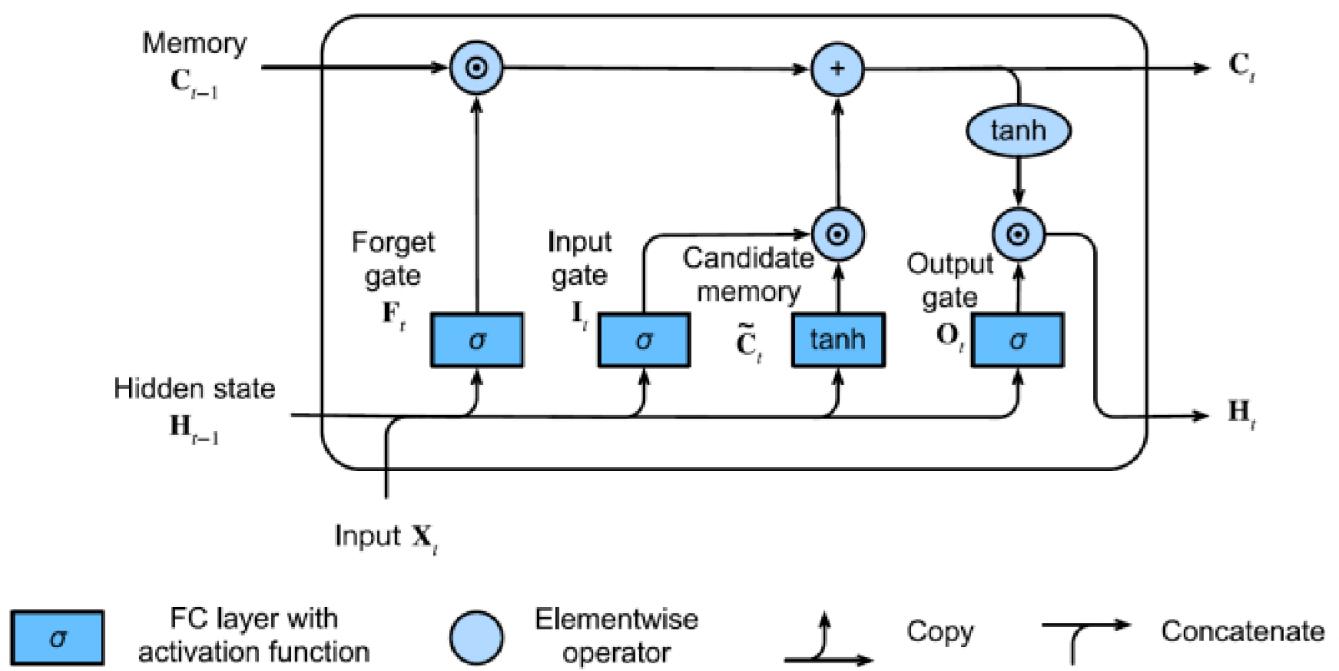


Figure 11: Computing the hidden state in an LSTM model.

## 4.5 Deep recurrent neural networks

- up to now, we only discussed RNNs with a single hidden layer
- in it, the specific functional form of how latent variables and observations interact is rather arbitrary
- this is not a big problem, as long as we have enough flexibility to model different types of interactions
- with a single layer, however, this can be quite challenging
- in the case of the linear models, we fixed this problem by adding more layers
- within RNNs, this is a bit trickier, since we first need to decide how and where to add extra nonlinearity

## 4.5 Deep recurrent neural networks

- in fact, we could stack multiple layers of RNNs on top of each other
- this results in a flexible mechanism, due to the combination of several simple layers
- in particular, data might be relevant at different levels of the stack
- for instance, we might want to keep high-level data about financial market conditions (bear or bull market) available, whereas, at a lower level, we only record shorter-term temporal dynamics
- beyond all the above abstract discussion, it is probably easiest to understand the family of models we are interested in by reviewing Figure 12; it describes a deep RNN with  $L$  hidden layers
- each hidden state is continuously passed to both the next time step of the current layer, and the current time step of the next layer

## 4.5 Deep recurrent neural networks

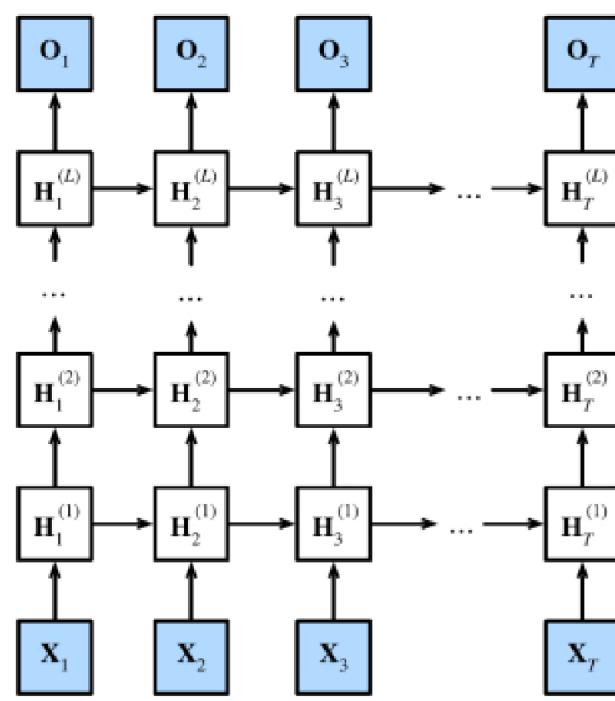


Figure 12: Architecture of a deep RNN.

## 4.5 Deep recurrent neural networks

- we can formalize the functional dependencies within the deep architecture of  $L$  hidden layers depicted in Figure 12
- our following discussion focuses primarily on the vanilla RNN model, but it applies to other sequence models, too
- suppose that we have a mini-batch input  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (number of examples:  $n$ , number of inputs in each example:  $d$ ) at time step  $t$
- at the same time step, let the hidden state of the  $l$ th hidden layer ( $l = 1, \dots, L$ ) be  $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$  (number of hidden units:  $h$ ) and the output layer variable be  $\mathbf{O}_t \in \mathbb{R}^{n \times q}$  (number of outputs:  $q$ )
- setting  $\mathbf{H}_t^{(0)} = \mathbf{X}_t$ , the hidden state of the  $l$ th hidden layer that uses the activation function  $\phi_l$  is expressed as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}), \quad (6)$$

where the weights  $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$  and  $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$ , together with the bias  $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$ , are the model parameters of the  $l$ th hidden layer

## 4.5 Deep recurrent neural networks

- in the end, the calculation of the output layer is only based on the hidden state of the final  $L$ th hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q,$$

where the weight  $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$  and the bias  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  are the model parameters of the output layer

- just as with MLPs, the number of hidden layers  $L$  and the number of hidden units  $h$  are hyperparameters
- in other words, they can be tuned or specified by us
- in addition, we can easily get a deep gated RNN by replacing the hidden state computation in (6) with that from a GRU or an LSTM

## 4.6 Sequence to sequence learning

- RNNs are used to design language models, which are key to natural language processing
- another important benchmark is *machine translation*, a central problem domain for *sequence transduction* models that transform input sequences into output sequences
- playing a crucial role in various modern AI applications, sequence transduction models will form the focus of the remainder of this chapter and Chapter 5
- thus, we introduce the machine translation problem

## 4.6 Sequence to sequence learning

- *machine translation* refers to the automatic translation of a sequence from one language to another
- for decades, statistical approaches had been dominant in this field, before the rise of end-to-end learning using neural networks
- the latter is often called *neural machine translation* to distinguish itself from *statistical machine translation* that involves statistical analysis in components such as the translation model and the language model
- we will focus on neural machine translation methods
- different from language models, for which the corpus is in one single language, machine translation datasets are composed of pairs of text sequences that are in the *source language* and the *target language*, respectively

## 4.6 Sequence to sequence learning

- thus, machine translation is a major problem domain for sequence transduction models, whose input and output are both variable-length sequences
- to handle this type of inputs and outputs, we can design an architecture with two major components
- the first component is an *encoder*: it takes a variable-length sequence as the input and transforms it into a state with a fixed shape
- the second component is a *decoder*: it maps the encoded state of a fixed shape to a variable-length sequence
- this is called an *encoder-decoder* architecture, which is depicted in Figure 13

## 4.6 Sequence to sequence learning

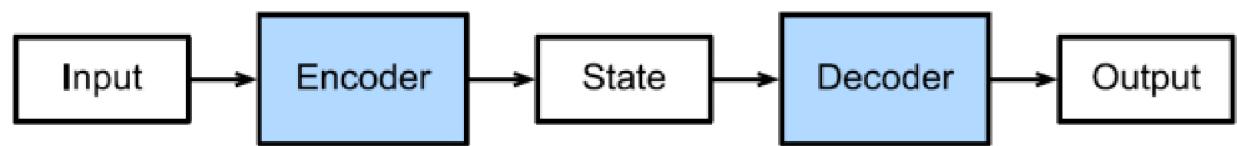


Figure 13: The encoder-decoder architecture.

## 4.6 Sequence to sequence learning

- let us take machine translation from English to French as an example
- given an input sequence in English: "They", "are", "watching", ".", this encoder-decoder architecture first encodes the variable-length input into a state, then decodes the state to generate the translated sequence token by token as the output: "Ils", "regardent", "
- the term "state" in the encoder-decoder architecture has probably inspired us to think about neural networks with states, when instantiating this architecture
- next, we will see how to apply RNNs to design sequence transduction models based on this encoder-decoder architecture

## 4.6 Sequence to sequence learning

- in machine translation, both the input and output are a variable-length sequence
- to address this type of problem, we have designed a general encoder-decoder architecture
- now, we will use two RNNs to design the encoder and the decoder of this architecture, and apply it to sequence to sequence learning for machine translation

## 4.6 Sequence to sequence learning

- following the design principle of the encoder-decoder architecture, the RNN encoder can take a variable-length sequence as the input, and transform it into a fixed-shape hidden state
- in other words, information of the input (source) sequence is *encoded* in the hidden state of the RNN encoder
- to generate the output sequence token by token, a separate RNN decoder can predict the next token based on what tokens have been seen (such as in language modeling) or generated, together with the encoded information of the input sequence
- Figure 14 illustrates how to use two RNNs for sequence to sequence learning in machine translation

## 4.6 Sequence to sequence learning

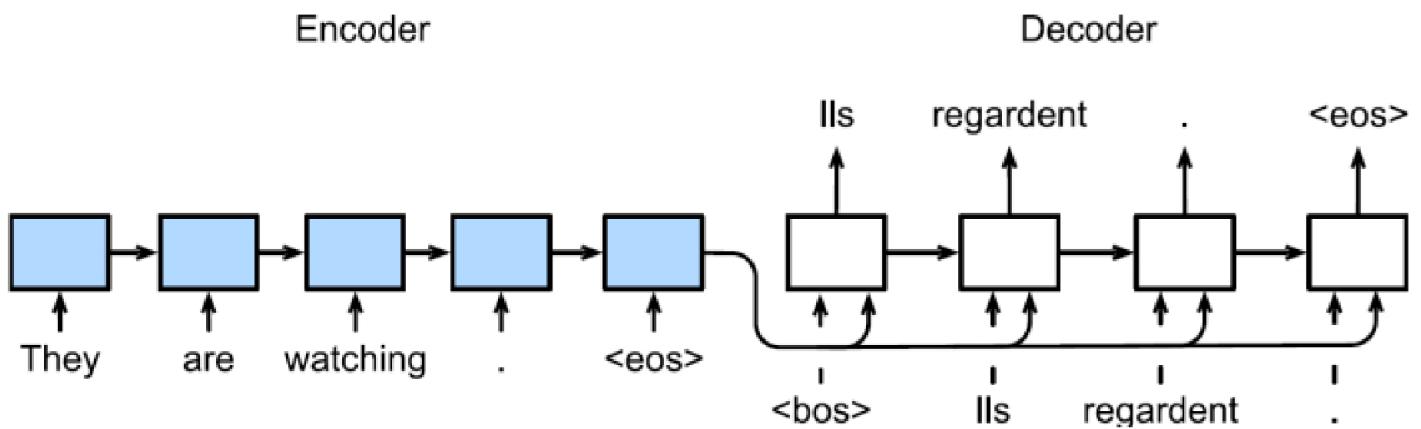


Figure 14: Sequence to sequence learning with an RNN encoder and an RNN decoder.

## 4.6 Sequence to sequence learning

- in Figure 14, the special “<eos>” token marks the end of the sequence
- the model can stop making predictions once this token is generated
- at the initial time step of the RNN decoder, there are two special design decisions
  - first, the special beginning-of-sequence “<bos>” token is an input
  - second, the final hidden state of the RNN encoder is used to initialize the hidden state of the decoder

## 4.6 Sequence to sequence learning

- in some designs, this is exactly how the encoded input sequence information is fed into the decoder for generating the output (target) sequence
- in some other designs, the final hidden state of the encoder is also fed into the decoder as part of the inputs at every time step, as shown in Figure 14
- similar to the training of language models, we can allow the labels to be the original output sequence, shifted by one token: "<bos>", "Ils", "regardent", "." → "Ils", "regardent", ".", "<eos>"
- in the following, we will explain the design of Figure 14 in greater detail

## 4.6 Sequence to sequence learning

- technically speaking, the encoder transforms an input sequence of variable length into a fixed-shape *context variable*  $\mathbf{c}$ , and encodes the input sequence information in this context variable
- as depicted in Figure 14, we can use an RNN to design the encoder
- let us consider a sequence example (batch size: 1)
- suppose that the input sequence is  $x_1, \dots, x_T$ , such that  $x_t$  is the  $t$ th token in the input text sequence
- at time step  $t$ , the RNN transforms the input feature vector  $\mathbf{x}_t$  for  $x_t$  and the hidden state  $\mathbf{h}_{t-1}$  from the previous time step into the current hidden state  $\mathbf{h}_t$
- we can use a function  $f$  to express the transformation of the RNN's recurrent layer:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

## 4.6 Sequence to sequence learning

- in general, the encoder transforms the hidden states at all the time steps into the context variable through a customized function  $q$ :

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T).$$

- for example, when choosing  $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$ , such as in Figure 14, the context variable is just the hidden state  $\mathbf{h}_T$  of the input sequence at the final time step
- so far, we have used an RNN to design the encoder, where a hidden state only depends on the input subsequence at, and before, the time step of the hidden state

## 4.6 Sequence to sequence learning

- as we just mentioned, the context variable  $\mathbf{c}$  of the encoder's output encodes the entire input sequence  $x_1, \dots, x_T$
- given the output sequence  $y_1, y_2, \dots, y_{T'}$  from the training dataset, for each time step  $t'$  (the symbol differs from the time step  $t$  of input sequences or encoders), the probability of the decoder output  $y_{t'}$  is conditional on the previous output subsequence  $y_1, \dots, y_{t'-1}$ , and the context variable  $\mathbf{c}$ , i.e.,  $P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c})$

## 4.6 Sequence to sequence learning

- to model this conditional probability on sequences, we can use another RNN as the decoder
- at any time step  $t'$  on the output sequence, the RNN takes the output  $y_{t'-1}$  from the previous time step and the context variable  $\mathbf{c}$  as its input, then transforms them and the previous hidden state  $\mathbf{s}_{t'-1}$  into the hidden state  $\mathbf{s}_{t'}$  at the current time step
- as a result, we can use a function  $g$  to express the transformation of the decoder's hidden layer:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}).$$

## 4.6 Sequence to sequence learning

- after obtaining the hidden state of the decoder, we can use an output layer and the softmax operation to compute the conditional probability distribution  $P(y_{t'}|y_1, \dots, y_{t'-1}, \mathbf{c})$  for the output at time step  $t'$
- to predict the output sequence token by token, at each decoder time step, the predicted token from the previous time step is fed into the decoder as an input
- similar to training, at the initial time step, the beginning-of-sequence (“<bos>”) token is fed into the decoder
- this prediction process is illustrated in Figure 15
- when the end-of-sequence (“<eos>”) token is predicted, the prediction of the output sequence is complete

## 4.6 Sequence to sequence learning

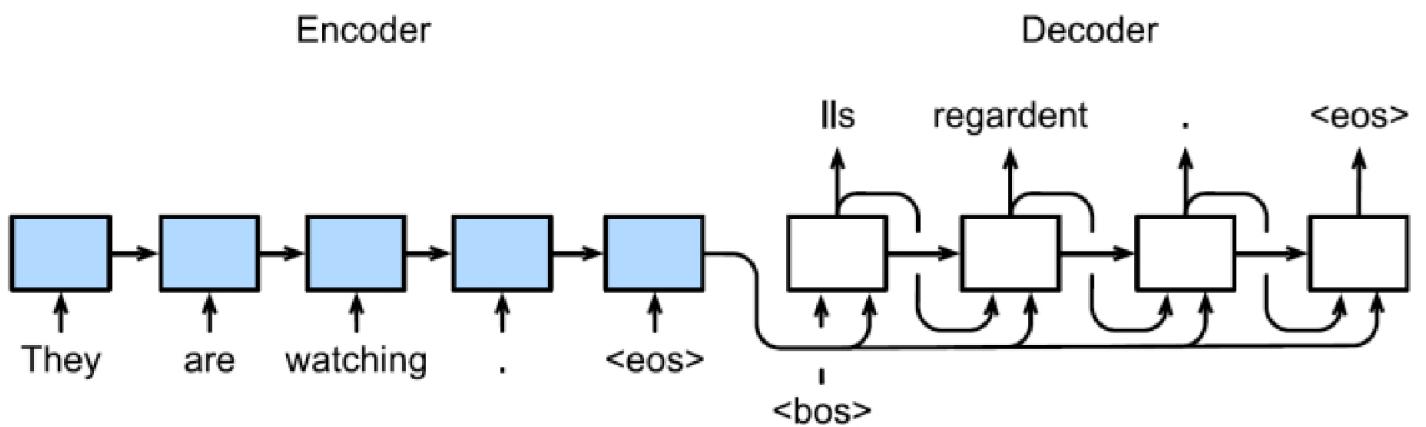


Figure 15: Predicting the output sequence token by token using an RNN encoder-decoder.

## 4.6 Sequence to sequence learning

- we can evaluate a predicted sequence by comparing it with the label sequence (the ground truth)
- *BLEU (Bilingual Evaluation Understudy)*, though originally proposed for evaluating machine translation results, has been extensively used in measuring the quality of output sequences for different applications
- in principle, for any  $n$ -grams in the predicted sequence, BLEU evaluates whether these  $n$ -grams appear in the label sequence

## 4.6 Sequence to sequence learning

- denote by  $p_n$  the precision of  $n$ -grams, which is the ratio of the number of matched  $n$ -grams in the predicted and label sequences to the number of  $n$ -grams in the predicted sequence
- to explain, given a label sequence  $A, B, C, D, E, F$ , and a predicted sequence  $A, B, B, C, D$ , we have  $p_1 = 4/5$ ,  $p_2 = 3/4$ ,  $p_3 = 1/3$ , and  $p_4 = 0$
- besides, let  $\text{len}_{\text{label}}$  and  $\text{len}_{\text{pred}}$  be the number of tokens in the label sequence and the predicted sequence, respectively
- then, BLEU is defined as:

$$\exp \left( \min \left( 0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}} \right) \right) \prod_{n=1}^k p_n^{1/2^n}, \quad (7)$$

where  $k$  is the longest  $n$ -grams for matching

## 4.6 Sequence to sequence learning

- based on the definition of BLEU in (7), whenever the predicted sequence is the same as the label sequence, BLEU is 1
- moreover, since matching longer  $n$ -grams is more difficult, BLEU assigns a greater weight to a longer  $n$ -gram precision
- specifically, when  $p_n$  is fixed,  $p_n^{1/2^n}$  increases as  $n$  grows (the original paper uses  $p_n^{1/n}$ )
- furthermore, since predicting shorter sequences tends to obtain a higher  $p_n$  value, the coefficient before the multiplication term in (7) penalizes shorter predicted sequences
- for example, when  $k = 2$ , given the label sequence  $A, B, C, D, E, F$  and the predicted sequence  $A, B$ , although  $p_1 = p_2 = 1$ , the penalty factor  $\exp(1 - 6/2) \approx 0.14$  lowers the BLEU

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 9

December 6th, 2022

## 5 Attention mechanisms

- the optic nerve of our visual system receives massive sensory input, far exceeding what the brain can fully process
- fortunately, not all stimuli are equal
- focalization and concentration of consciousness have enabled us to direct attention to objects of interest, such as preys and predators, in the complex visual environment
- the ability of paying attention to only a small fraction of the information has evolutionary significance, allowing human beings to live and succeed

## 5.1 Attention cues

- information in our environment is not scarce, attention is
- when inspecting a visual scene, our optic nerve receives information at the order of  $10^8$  bits per second, far exceeding what our brain can fully process
- fortunately, we learned from experience (also known as data) that *not all sensory inputs are equal*
- throughout human history, the capability of directing attention to only a fraction of information of interest has enabled our brain to allocate resources more smartly to survive, to grow, and to socialize, such as detecting predators, preys, and mates

## 5.1 Attention cues

- to explain how our attention is deployed in the visual world, a two-component framework has emerged
- in this framework, subjects selectively direct the spotlight of attention using both the *nonvolitional cue* and the *volitional cue*
- the nonvolitional cue is based on the *saliency* of objects in the environment
- imagine there are five objects in front of us: a newspaper, a research paper, a cup of coffee, a notebook, and a book, such as in Figure 1
- while all the paper products are printed in black and white, the coffee cup is red
- in other words, this coffee is intrinsically salient in this visual environment, automatically and involuntarily drawing attention
- so, we bring the fovea (the center of the macula, where visual acuity is highest) onto the coffee, as shown in Figure 1

## 5.1 Attention cues

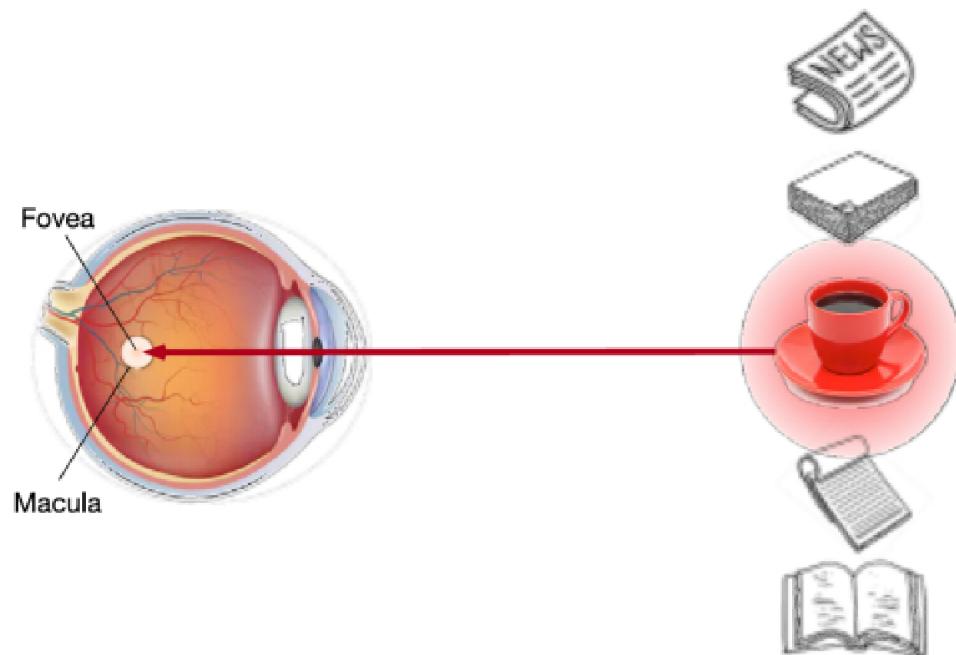


Figure 1: Using the nonvolitional cue based on saliency (red cup, non-paper), attention is involuntarily directed to the coffee.

## 5.1 Attention cues

- after drinking coffee, we become caffeinated, and want to read a book
- so we turn our head, refocus our eyes, and look at the book, as depicted in Figure 2
- different from the case in Figure 1, where the coffee biases us towards selecting based on saliency, in this task-dependent case, we select the book under cognitive and volitional control
- using the volitional cue based on variable selection criteria, this form of attention is more deliberate
- it is also more powerful, with the subject's voluntary effort

## 5.1 Attention cues

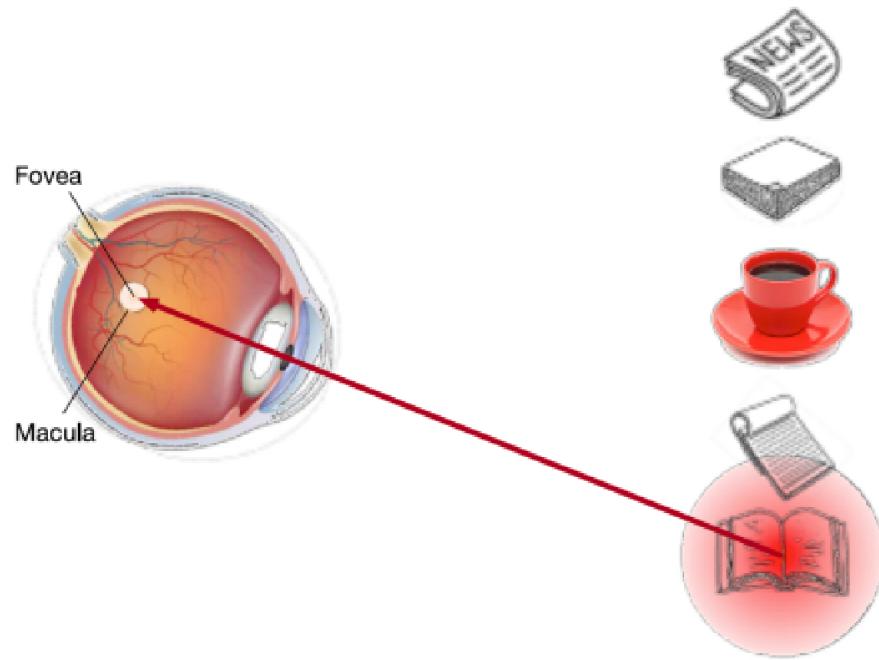


Figure 2: Using the volitional cue (want to read a book) that is task-dependent, attention is directed to the book under volitional control.

## 5.1 Attention cues

- inspired by the nonvolitional and volitional attention cues that explain the attentional deployment, in the following we will describe a framework for designing attention mechanisms by incorporating these two attention cues
- to begin with, consider the simpler case where only nonvolitional cues are available
- to bias selection over sensory inputs, we can simply use a parameterized fully-connected layer or even a non-parameterized max or average pooling layer

## 5.1 Attention cues

- therefore, what sets attention mechanisms apart from those fully-connected layers or pooling layers is the inclusion of the volitional cues
- in the context of attention mechanisms, we refer to volitional cues as *queries*
- given any query, attention mechanisms bias selection over sensory inputs (e.g., intermediate feature representations) via *attention pooling*
- these sensory inputs are called *values*, in the context of attention mechanisms
- more generally, every value is paired with a *key*, which can be thought of as the nonvolitional cue of that sensory input
- as shown in Figure 3, we can design attention pooling so that the given query (volitional cue) can interact with keys (nonvolitional cues), which guides bias selection over values (sensory inputs)

## 5.1 Attention cues

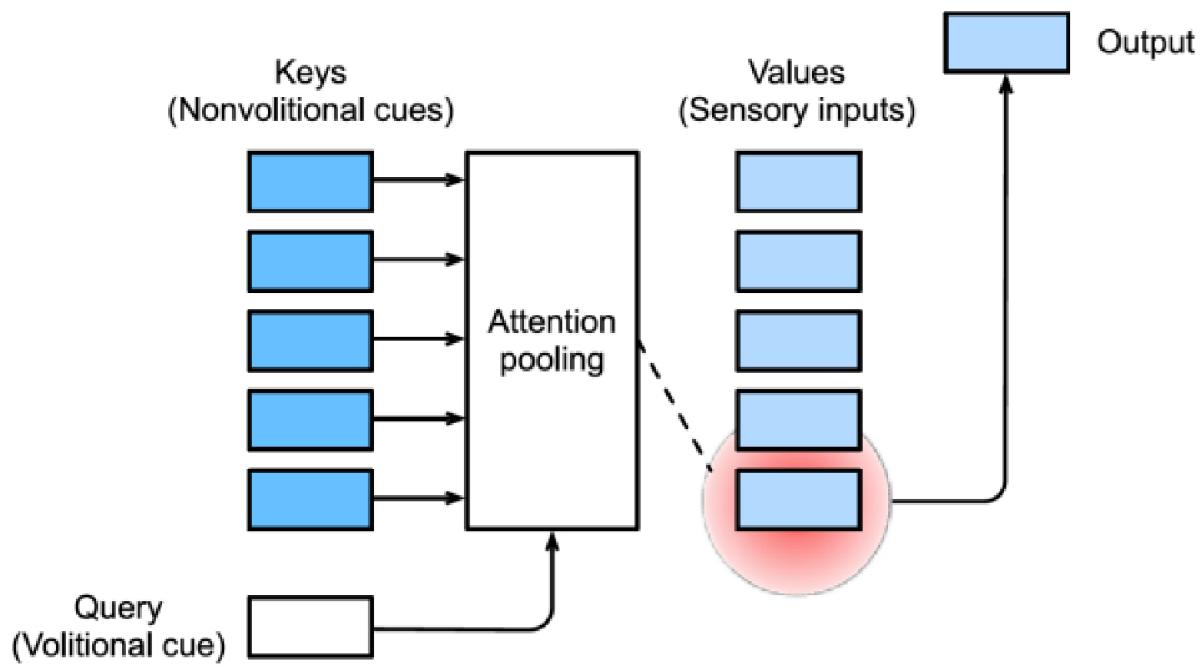


Figure 3: Attention mechanisms bias selection over values (sensory inputs) via attention pooling, which incorporates queries (volitional cues) and keys (nonvolitional cues).

## 5.2 Attention pooling: Nadaraya-Watson kernel regression

- now we know the major components of attention mechanisms under the framework in Figure 3
- to recapitulate, the interactions between queries (volitional cues) and keys (nonvolitional cues) result in *attention pooling*
- the attention pooling selectively aggregates values (sensory inputs) to produce the output
- in this section, we will describe attention pooling in greater detail, to give a high-level view of how attention mechanisms work in practice
- specifically, the Nadaraya-Watson kernel regression model, proposed in 1964, is a simple yet complete example for demonstrating machine learning with attention mechanisms

## 5.2 Attention pooling: Nadaraya-Watson kernel regression

- to keep things simple, let us consider the following regression problem: given a dataset of input-output pairs  $\{(x_1, y_1), \dots, (x_n, y_n)\}$ , how to learn  $f$  to predict the output  $\hat{y} = f(x)$  for any new input  $x$ ?
- we begin with perhaps the world's "dumbest" estimator for this regression problem: using average pooling to average over all the training outputs:

$$f(x) = \frac{1}{n} \sum_{i=1}^n y_i. \quad (1)$$

- obviously, average pooling omits the inputs  $x_i$
- a better idea was proposed by Nadaraya and Watson, namely to weigh the outputs  $y_i$  according to their input locations:

$$f(x) = \sum_{i=1}^n \frac{K(x - x_i)}{\sum_{j=1}^n K(x - x_j)} y_i, \quad (2)$$

where  $K$  is a *kernel*

- the estimator in (2) is called *Nadaraya-Watson kernel regression*; here, we will not dive into details of kernels

## 5.2 Attention pooling: Nadaraya-Watson kernel regression

- recall the framework of attention mechanisms in Figure 3; from the perspective of attention, we can rewrite (2) in a more generalized form of *attention pooling*:

$$f(x) = \sum_{i=1}^n \alpha(x, x_i) y_i, \quad (3)$$

where  $x$  is the *query* and  $(x_i, y_i)$  is the *key-value* pair

- comparing (3) and (1), the attention pooling here is a weighted average of values  $y_i$
- the *attention weight*  $\alpha(x, x_i)$  in (3) is assigned to the corresponding value  $y_i$  based on the interaction between the query  $x$  and the key  $x_i$ , modeled by  $\alpha$
- for any query, its attention weights over all the key-value pairs are a valid probability distribution: they are non-negative and sum up to one

## 5.2 Attention pooling: Nadaraya-Watson kernel regression

- to gain intuitions of attention pooling, we can consider a Gaussian kernel defined as:

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right).$$

- plugging the Gaussian kernel into (3) and (2) gives:

$$\begin{aligned} f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\ &= \sum_{i=1}^n \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^n \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^n \text{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \tag{4}$$

- in (4), a key  $x_i$  that is closer to the given query  $x$  will get *more attention* via a *larger attention weight* assigned to the key's corresponding value  $y_i$

## 5.2 Attention pooling: Nadaraya-Watson kernel regression

- notably, Nadaraya-Watson kernel regression is a nonparametric model; thus (4) is an example of *nonparametric attention pooling*
- nonparametric Nadaraya-Watson kernel regression enjoys the *consistency* benefit: given enough data, this model converges to the optimal solution
- nonetheless, we can easily integrate learnable parameters into attention pooling
- as an example, slightly different from (4), in the following, the distance between the query  $x$  and the key  $x_i$  is multiplied by a learnable parameter  $w$ :

$$\begin{aligned} f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\ &= \sum_{i=1}^n \frac{\exp(-\frac{1}{2}((x - x_i)w)^2)}{\sum_{j=1}^n \exp(-\frac{1}{2}((x - x_j)w)^2)} y_i \\ &= \sum_{i=1}^n \text{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i. \end{aligned}$$

## 5.3 Attention scoring functions

- in the previous section, we used a Gaussian kernel to model interactions between queries and keys
- treating the exponent of the Gaussian kernel in (4) as an *attention scoring function* (or *scoring function*, for short), the results of this function were essentially fed into a softmax operation
- as a result, we obtained a probability distribution (attention weights) over values that are paired with keys
- in the end, the output of the attention pooling is simply a weighted sum of the values based on these attention weights

## 5.3 Attention scoring functions

- at a high level, we can use the above algorithm to instantiate the framework of attention mechanisms in Figure 3
- denoting an attention scoring function by  $a$ , Figure 4 illustrates how the output of attention pooling can be computed as a weighted sum of values
- since attention weights are a probability distribution, the weighted sum is essentially a weighted average

## 5.3 Attention scoring functions

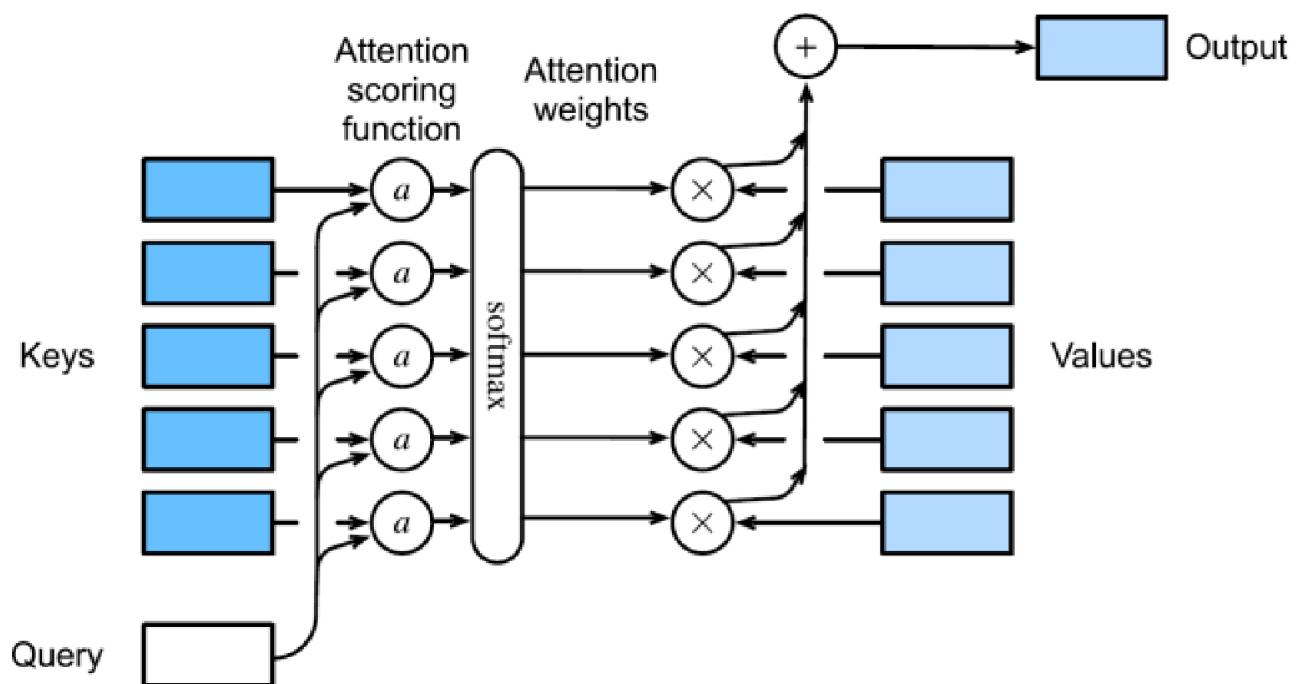


Figure 4: Computing the output of attention pooling as a weighted average of values.

## 5.3 Attention scoring functions

- mathematically, suppose that we have a query  $\mathbf{q} \in \mathbb{R}^q$  and  $m$  key-value pairs  $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)$ , where any  $\mathbf{k}_i \in \mathbb{R}^k$  and any  $\mathbf{v}_i \in \mathbb{R}^v$
- the attention pooling  $f$  is instantiated as a weighted sum of the values:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i \in \mathbb{R}^v, \quad (5)$$

where the attention weight (scalar) for the query  $\mathbf{q}$  and key  $\mathbf{k}_i$  is computed by the softmax operation of an attention scoring function  $a$  that maps two vectors to a scalar:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^m \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \quad (6)$$

- as we can see, different choices of the attention scoring function  $a$  lead to different behaviors of attention pooling
- in this section, we introduce two popular scoring functions that we will use to develop more sophisticated attention mechanisms later

## 5.3 Attention scoring functions

- in general, when queries and keys are vectors of different lengths, we can use additive attention as the scoring function
- given a query  $\mathbf{q} \in \mathbb{R}^q$  and a key  $\mathbf{k} \in \mathbb{R}^k$ , the *additive attention* scoring function is defined as:

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \quad (7)$$

where the parameters  $\mathbf{W}_q \in \mathbb{R}^{h \times q}$ ,  $\mathbf{W}_k \in \mathbb{R}^{h \times k}$ , and  $\mathbf{w}_v \in \mathbb{R}^h$  are learnable

- equivalent to (7), the query and the key are concatenated and fed into an MLP with a single hidden layer whose number of hidden units is  $h$ , a hyperparameter, and is implemented by using tanh as the activation function and disabling bias terms

## 5.3 Attention scoring functions

- a more computationally efficient design for the scoring function can be simply the *dot product*
- however, the dot product operation requires that both the query and the key have the same vector length, say  $d$
- assume that all the elements of the query and the key are independent random variables with zero mean and unit variance
- the dot product of both vectors has zero mean and a variance of  $d$

## 5.3 Attention scoring functions

- to ensure that the variance of the dot product still remains one, regardless of vector length, the *scaled dot-product attention* scoring function:

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d}$$

divides the dot product by  $\sqrt{d}$

- in practice, we often think in mini-batches for efficiency, such as computing attention for  $n$  queries and  $m$  key-value pairs, where queries and keys are of length  $d$ , and values are of length  $v$
- the scaled dot-product attention of queries  $\mathbf{Q} \in \mathbb{R}^{n \times d}$ , keys  $\mathbf{K} \in \mathbb{R}^{m \times d}$ , and values  $\mathbf{V} \in \mathbb{R}^{m \times v}$  is:

$$\text{softmax} \left( \frac{\mathbf{Q}^\top \mathbf{K}}{\sqrt{d}} \right) \mathbf{V} \in \mathbb{R}^{n \times v}. \quad (8)$$

## 5.4 Bahdanau attention

- we studied the machine translation problem in Section 4.6, where we designed an encoder-decoder architecture, based on two RNNs, for sequence to sequence learning
- specifically, the RNN encoder transforms a variable-length sequence into a fixed-shape context variable, then the RNN decoder generates the output (target) sequence token by token, based on the generated tokens and the context variable
- however, even though not all the input (source) tokens are useful for decoding a certain token, the *same* context variable that encodes the entire input sequence is still used at each decoding step

## 5.4 Bahdanau attention

- in a separate but related challenge of handwriting generation for a given text sequence, Graves designed a differentiable attention model to align text characters with the much longer pen trace, where the alignment moves only in one direction
- inspired by the idea of learning to align, Bahdanau et al. proposed a differentiable attention model without the severe unidirectional alignment limitation
- when predicting a token, if not all the input tokens are relevant, the model attends (or attends) only to parts of the input sequence that are relevant to the current prediction
- this is achieved by treating the context variable as an output of attention pooling

## 5.4 Bahdanau attention

- when describing Bahdanau attention for the RNN encoder-decoder below, we will follow the same notation in Section 4.6
- the new attention-based model is the same as that in Section 4.6, except that the context variable  $\mathbf{c}$  is replaced by  $\mathbf{c}_{t'}$  at any decoding time step  $t'$
- suppose that there are  $T$  tokens in the input sequence, then the context variable at the decoding time step  $t'$  is the output of attention pooling:

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t) \mathbf{h}_t,$$

where the decoder hidden state  $\mathbf{s}_{t'-1}$  at time step  $t' - 1$  is the query, and the encoder hidden states  $\mathbf{h}_t$  are both the keys and values, and the attention weight  $\alpha$  is computed as in (6), using the additive attention scoring function defined by (7)

- slightly different from the vanilla RNN encoder-decoder architecture, the same architecture with Bahdanau attention is depicted in Figure 5

## 5.4 Bahdanau attention

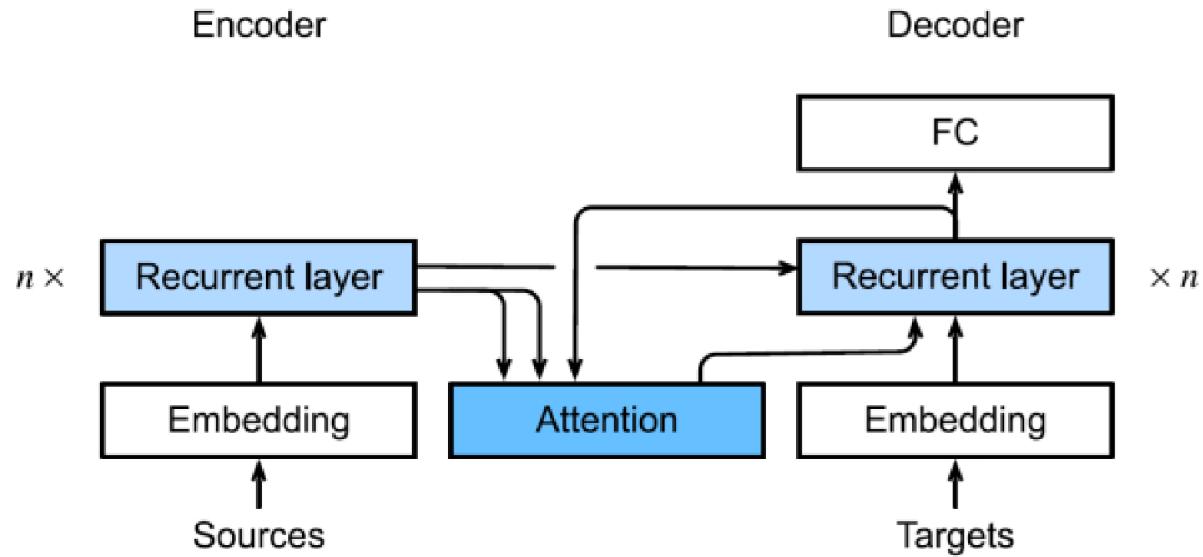


Figure 5: Layers in an RNN encoder-decoder model with Bahdanau attention.

# Thank you!



# Image Processing and Recognition

Dr. Călin-Adrian POPA

## Lecture 10

December 13th, 2022

- in practice, given the same set of queries, keys, and values, we may want our model to combine knowledge from different behaviors of the same attention mechanism, such as capturing dependencies of various ranges (e.g., shorter-range vs. longer-range) within a sequence
- thus, it may be beneficial to allow our attention mechanism to jointly use different representation subspaces of queries, keys, and values

## 5.5 Multi-head attention

- to this end, instead of performing a single attention pooling, queries, keys, and values can be transformed with  $h$  independently learned linear projections
- then, these  $h$  projected queries, keys, and values are fed into attention pooling in parallel
- in the end, the  $h$  attention pooling outputs are concatenated and transformed with another learned linear projection, to produce the final output
- this design is called *multi-head attention*, where each of the  $h$  attention pooling outputs is a *head*
- using fully-connected layers to perform learnable linear transformations, Figure 6 describes multi-head attention

## 5.5 Multi-head attention

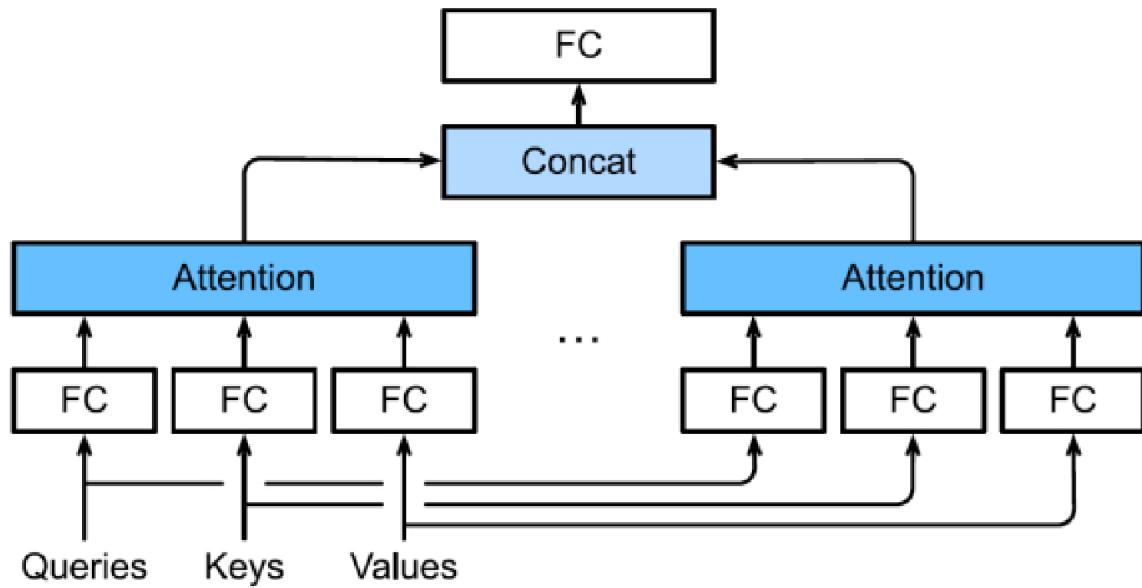


Figure 6: Multi-head attention, where multiple heads are concatenated, then linearly transformed.

## 5.5 Multi-head attention

- let us formalize this model mathematically; given a query  $\mathbf{q} \in \mathbb{R}^{d_q}$ , a key  $\mathbf{k} \in \mathbb{R}^{d_k}$ , and a value  $\mathbf{v} \in \mathbb{R}^{d_v}$ , each attention head  $\mathbf{h}_i$  ( $i = 1, \dots, h$ ) is computed as:

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)} \mathbf{q}, \mathbf{W}_i^{(k)} \mathbf{k}, \mathbf{W}_i^{(v)} \mathbf{v}) \in \mathbb{R}^{p_v},$$

where the parameters  $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$ ,  $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ , and  $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$  are learnable, and  $f$  is attention pooling, such as additive attention or scaled dot-product attention, discussed in Section 5.3

- the multi-head attention output is another linear transformation via learnable parameters  $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$  of the concatenation of the  $h$  heads:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}.$$

- based on this design, each head may attend to different parts of the input; more sophisticated functions than the simple weighted average can be expressed

## 5.6 Self-attention and positional encoding

- in deep learning, we often use CNNs or RNNs to encode a sequence
- now, with attention mechanisms, imagine that we feed a sequence of tokens into attention pooling, so that the same set of tokens act as queries, keys, and values
- specifically, each query attends to all the key-value pairs and generates one attention output
- since the queries, keys, and values come from the same place, this performs *self-attention*, which is also called *intra-attention*
- in this section, we will discuss sequence encoding using self-attention, including using additional information for the sequence order

## 5.6 Self-attention and positional encoding

- given a sequence of input tokens  $\mathbf{x}_1, \dots, \mathbf{x}_n$ , where any  $\mathbf{x}_i \in \mathbb{R}^d$  ( $1 \leq i \leq n$ ), its self-attention outputs a sequence of the same length  $\mathbf{y}_1, \dots, \mathbf{y}_n$ , where:

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \dots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d,$$

according to the definition of attention pooling  $f$  in (5)

- let us compare architectures for mapping a sequence of  $n$  tokens to another sequence of equal length, where each input or output token is represented by a  $d$ -dimensional vector
- specifically, we will consider CNNs, RNNs, and self-attention
- we will compare their computational complexity, sequential operations, and maximum path lengths
- note that sequential operations prevent parallel computation, while a shorter path between any combination of sequence positions makes it easier to learn long-range dependencies within the sequence

## 5.6 Self-attention and positional encoding

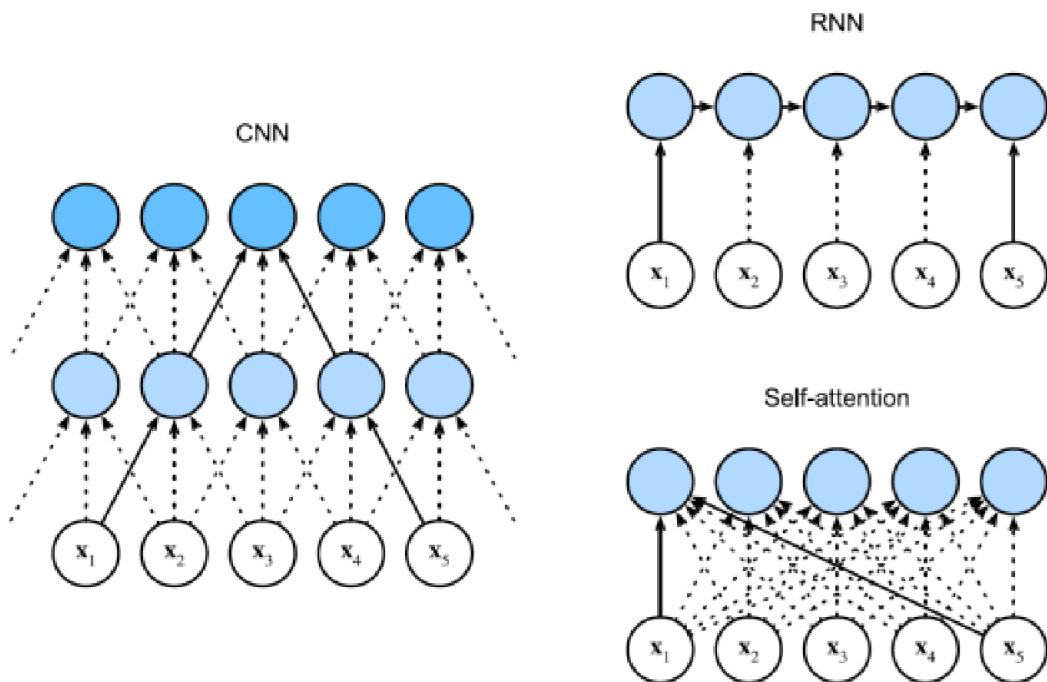


Figure 7: Comparing CNN (padding tokens are omitted), RNN, and self-attention architectures.

## 5.6 Self-attention and positional encoding

- consider a convolutional layer whose kernel size is  $k$
- since the sequence length is  $n$ , the numbers of input and output channels are both  $d$ , the computational complexity of the convolutional layer is  $\mathcal{O}(knd^2)$
- as Figure 7 shows, CNNs are hierarchical, so there are  $\mathcal{O}(1)$  sequential operations and the maximum path length is  $\mathcal{O}(n/k)$
- for example,  $x_1$  and  $x_5$  are within the receptive field of a two-layer CNN with kernel size 3 in Figure 7

## 5.6 Self-attention and positional encoding

- when updating the hidden state of RNNs, multiplication of the  $d \times d$  weight matrix and the  $d$ -dimensional hidden state has a computational complexity of  $\mathcal{O}(d^2)$
- since the sequence length is  $n$ , the computational complexity of the recurrent layer is  $\mathcal{O}(nd^2)$
- according to Figure 7, there are  $\mathcal{O}(n)$  sequential operations that cannot be parallelized and the maximum path length is also  $\mathcal{O}(n)$

## 5.6 Self-attention and positional encoding

- in self-attention, the queries, keys, and values are all  $n \times d$  matrices
- consider the scaled dot-product attention in (8), where a  $n \times d$  matrix is multiplied by a  $d \times n$  matrix, then the output  $n \times n$  matrix is multiplied by a  $n \times d$  matrix
- as a result, the self-attention has a  $\mathcal{O}(n^2d)$  computational complexity
- as we can see in Figure 7, each token is directly connected to any other token via self-attention
- therefore, computation can be parallel with  $\mathcal{O}(1)$  sequential operations and the maximum path length is also  $\mathcal{O}(1)$

## 5.6 Self-attention and positional encoding

- all in all, both CNNs and self-attention enjoy parallel computation, and self-attention has the shortest maximum path length
- however, the quadratic computational complexity with respect to the sequence length makes self-attention prohibitively slow for very long sequences
- unlike RNNs that recurrently process tokens of a sequence one by one, self-attention ditches sequential operations in favor of parallel computation
- to use the sequence order information, we can inject absolute or relative positional information by adding *positional encoding* to the input representations
- positional encodings can be either learned or fixed; in the following, we describe a fixed positional encoding based on sine and cosine functions

## 5.6 Self-attention and positional encoding

- suppose that the input representation  $\mathbf{X} \in \mathbb{R}^{n \times d}$  contains the  $d$ -dimensional embeddings for  $n$  tokens of a sequence
- the positional encoding outputs  $\mathbf{X} + \mathbf{P}$  using a positional embedding matrix  $\mathbf{P} \in \mathbb{R}^{n \times d}$  of the same shape, whose element on the  $i$ th row and the  $(2j)$ th or the  $(2j+1)$ th column is:

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{9}$$

## 5.6 Self-attention and positional encoding

- in the positional embedding matrix  $P$ , rows correspond to positions within a sequence and columns represent different positional encoding dimensions
- in the example in Figure 8, we can see that the 6th and the 7th columns of the positional embedding matrix have a higher frequency than the 8th and the 9th columns
- the offset between the 6th and the 7th (same for the 8th and the 9th) columns is due to the alternation of sine and cosine functions

## 5.6 Self-attention and positional encoding

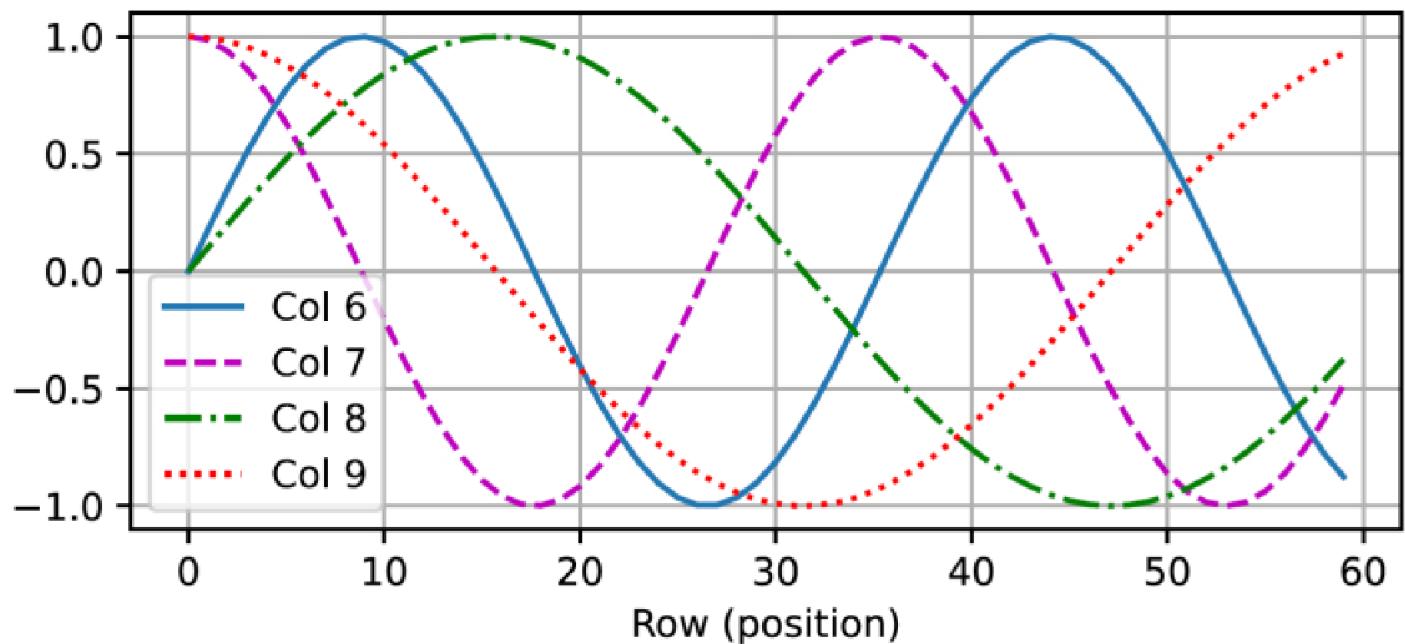


Figure 8: Positional encoding example.

## 5.6 Self-attention and positional encoding

- besides capturing absolute positional information, the above positional encoding also allows a model to easily learn to attend by relative positions
- this is because, for any fixed position offset  $\delta$ , the positional encoding at position  $i + \delta$  can be represented by a linear projection of that at position  $i$
- this projection can be explained mathematically; denoting  $\omega_j = 1/10000^{2j/d}$ , any pair of  $(p_{i,2j}, p_{i,2j+1})$  in (9) can be linearly projected to  $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ , for any fixed offset  $\delta$ :

$$\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} = \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\ = \begin{bmatrix} \sin((i + \delta)\omega_j) \\ \cos((i + \delta)\omega_j) \end{bmatrix} \\ = \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},$$

where the  $2 \times 2$  projection matrix does not depend on any position index  $i$

## 5.7 Transformer

- we have compared CNNs, RNNs, and self-attention in Section 5.6
- notably, self-attention enjoys both parallel computation and the shortest maximum path length
- therefore, naturally, it is appealing to design deep architectures by using self-attention
- unlike earlier self-attention models that still rely on RNNs for input representations, the *transformer* model is solely based on attention mechanisms, without any convolutional or recurrent layer

## 5.7 Transformer

- though originally proposed for sequence to sequence learning on text data, transformers have been pervasive in a wide range of modern deep learning applications, such as in areas of language, vision, speech, and reinforcement learning
- as an instance of the encoder-decoder architecture, the overall architecture of the transformer is presented in Figure 9
- as we can see, the transformer is composed of an encoder and a decoder
- different from Bahdanau attention for sequence to sequence learning in Figure 5, the input (source) and output (target) sequence embeddings are added with positional encoding, before being fed into the encoder and the decoder that stack modules based on self-attention

## 5.7 Transformer

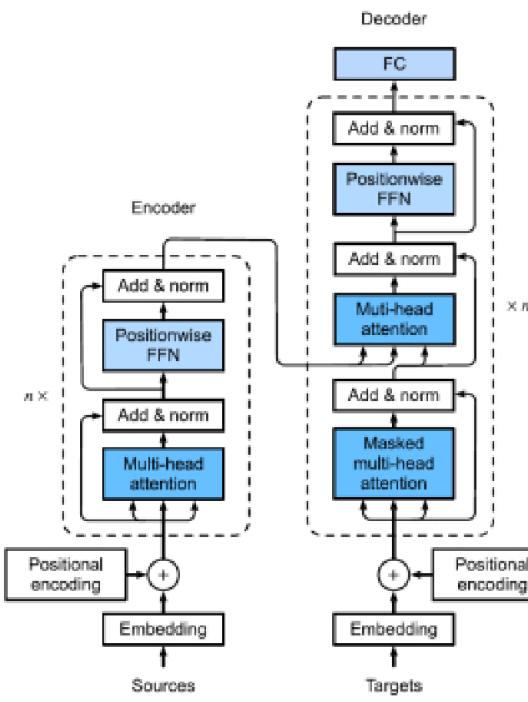


Figure 9: The transformer architecture.

## 5.7 Transformer

- now, we provide an overview of the transformer architecture in Figure 9
- on a high level, the transformer encoder is a stack of multiple identical layers, where each layer has two sublayers (either is denoted as sublayer)
- the first is a multi-head self-attention pooling and the second is a position-wise feed-forward network
- specifically, in the encoder self-attention, queries, keys, and values are all from the outputs of the previous encoder layer

## 5.7 Transformer

- inspired by the ResNet design in Section 3.10, a residual connection is employed around both sublayers
- in the transformer, for any input  $\mathbf{x} \in \mathbb{R}^d$ , at any position of the sequence, we require that  $\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ , so that the residual connection  $\mathbf{x} + \text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$  is feasible
- this addition from the residual connection is immediately followed by *layer normalization*
- as a result, the transformer encoder outputs a  $d$ -dimensional vector representation for each position of the input sequence

## 5.7 Transformer

- the transformer decoder is also a stack of multiple identical layers with residual connections and layer normalizations
- besides the two sublayers described in the encoder, the decoder inserts a third sublayer, known as the *encoder-decoder attention*, between these two
- in the encoder-decoder attention, queries are from the outputs of the previous decoder layer, and the keys and values are from the transformer encoder outputs
- in the decoder self-attention, queries, keys, and values are all from the outputs of the previous decoder layer
- however, each position in the decoder is allowed to only attend to all positions in the decoder up to that position
- this *masked* attention preserves the auto-regressive property, ensuring that the prediction only depends on those output tokens that have been generated

## 5.7 Transformer

- the position-wise feed-forward network transforms the representation at all the sequence positions using the same MLP; this is why we call it *position-wise*
- now, let us focus on the “add & norm” component in Figure 9
- as we described at the beginning of this section, this is a residual connection immediately followed by layer normalization; both are key to effective deep architectures
- in Section 3.9, we explained how batch normalization recenters and rescales across the examples within a mini-batch
- layer normalization is the same as batch normalization, except that the former normalizes across the feature dimension
- despite its pervasive applications in computer vision, batch normalization is usually empirically less effective than layer normalization in natural language processing tasks, whose inputs are often variable-length sequences

## 5.7 Transformer

- when training sequence-to-sequence models, tokens at all the positions (time steps) of the output sequence are known
- however, during prediction, the output sequence is generated token by token; thus, at any decoder time step, only the generated tokens can be used in the decoder self-attention
- to preserve auto-regression in the decoder, its masked self-attention specifies that any query should only attend to all positions in the decoder up to the query position
- although the transformer architecture was originally proposed for sequence-to-sequence learning, either the transformer encoder or the transformer decoder is often individually used for different deep learning tasks

# Thank you!

