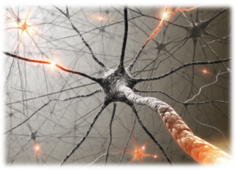
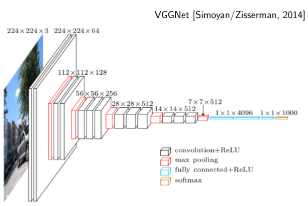
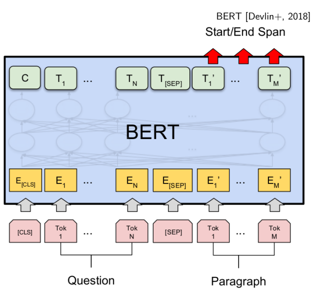
Machine learning: differentiable programming



- In this module, I'll briefly introduce the idea of differentiable programming, which runs with the ideas of computation graphs and backpropagation that we developed for simple neural networks.
- There is enough to say here to fill up an entire course, so I will keep things high-level but try to highlight the power of **composition**.
- Aside: Differentiable programming is closely related to deep learning. I've adopted the former term as a more precise way to highlight the mechanics of writing models as you would write code.

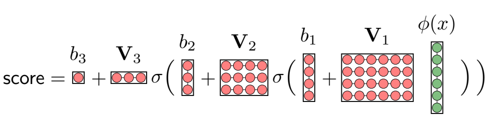
Deep learning models

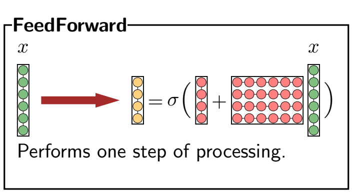




- If you look around at deep learning today, there are some pretty complex models which have many layers, attention mechanisms, residual connections, layerwise normalization, to name a few, which might be overwhelming at first glance.
- However, if you look closer, these complex models are actually composed of functions, which themselves are composed of even simpler functions.
- This is the "programming" part of differentiable programming, which allows you to build up increasingly more powerful and sophisticated models without losing track of what's going on.

Feedforward neural networks

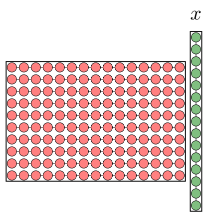




$$\text{score} = \textbf{FeedForward}(\textbf{FeedForward}(\textbf{FeedForward}(\phi(x))) = \textbf{FeedForward}^3(\phi(x))$$

- Let's revisit our familiar example, the three-layer neural network. In this model, we start with the feature vector $\phi(x)$ apply some operations (left-multiply by a matrix, add a bias term) to get the score.
- (Note that we highlight that each of these matrices should be interpreted as a collection of rows.)
- The main differences from before are that we've added bias terms and renamed the final weight vector $\mathbf{w}$ to $\mathbf{V}_3$ for consistency with the other layers.
- Let us now factor out a function of this model and call it **FeedForward**. This is a function that takes a fixed-dimensional vector x and produces another vector (with potentially another dimensionality). This function is implemented by multiplying by a matrix and applying an activation function (e.g., ReLU).
- Now we can not worry too much about the implementation, and just think of this as performing one step of processing. Compared to normal programming, this is admittedly vague, because the parameters (e.g., the red quantities) are not specified but rather learned from data. So differentiable programming requires us to necessarily be a bit loose.
- Using this brand new function, we can rewrite the three-layer neural network as follows. Strictly speaking, when we write **FeedForward**, we mean a function that has its own private parameters that need to be set via backpropagation.
- Another simplification is that we are eliding the input and output dimensionality (6 to 4, 4 to 3, and 3 to 1 in this example). These are hyperparameters which need to be set before learning.

Representing images

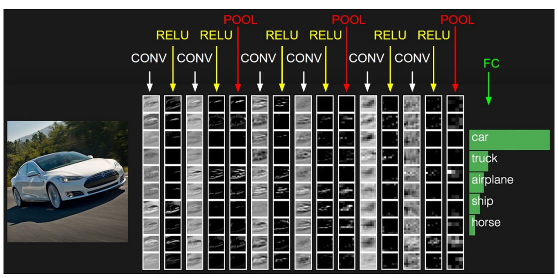


Problems:

- Matrix is huge (depending on resolution of image)
- Does not capture the spatial structure (locality) of images

- Now suppose we want to do image classification. We need to have some way of representing images.
- The **FeedForward** function takes in a vector as input, and we can represent an image as a long vector containing all the pixels (say, by concatenating all the rows).
- But then we would then have to have a huge matrix to transform this input, resulting in a lot of parameters (especially if the image is high resolution), which might be difficult to learn.
- The issue is that we're not leverage knowledge that these are images. If you permute the components of the input vector x and re-train, you will just get a permuted parameters, but the same predictions.

Convolutional neural networks



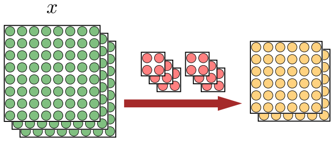
[Andrej Karpathy's demo]

- Convolutional neural networks (ConvNets or CNNs) is a refinement of the vanilla fully-connected neural networks tailored for images.
- It is also used for sequences such as text (everything is just 1D instead of 2D) or video (everything is 3D instead of 2D).
- Here is a visualization of a ConvNet making a prediction on this car image. There are a sequence of layers which turn the image into something increasingly abstract, and finally we get a vector representing the probabilities of the different object categories.
- You can click on this link to see Andrej Karpathy's demo, where you can create and train ConvNets in your browser.

Convolutional neural networks

Conv

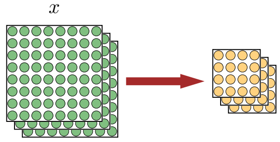
x



Detect local patterns with filters.

MaxPool

x



Take the max over each 2x2 region.

[Andrej Karpathy's demo]

$$\textbf{AlexNet}(x) = \textbf{FeedForward}^3(\textbf{MaxPool}(\textbf{Conv}^3(\textbf{MaxPool}(\textbf{Conv}(\textbf{MaxPool}(\textbf{Conv}(x)))))))$$

- So let us now define the two basic building blocks of ConvNets. We're not going to go through the details, but simply focus on the interface. (You should take CS231N if you want to learn more about ConvNets.)
- First, **Conv** takes as input an image, which can be represented as a **volume**, which is a matrix for each channel (red, green, blue), and each matrix has same height and width as the image.
- This function produces another volume whose height and width are either the same or a bit smaller than that of the input volume, and the number of output channels could be different.
- The output volume is constructed by sweeping a **filter** over the input volume, and taking the dot product of the filter with a local part of the input volume. That produces a number which you write into the output volume. The number of filters determines the number of channels in the output volume.
- The second operation is **MaxPool**, which simply takes an input volume and reduces the width and height by taking the max over local regions.
- Note that MaxPool does not have any parameters and has the same number of input channels as output channels.
- Given just these two functions, **Conv** and **MaxPool**, as well as our friend **FeedForward**, we can express the famous AlexNet architecture, which won the ImageNet competition in 2012 and arguably kicked off deep learning revolution.
- Note that each of these functions has its own parameters that need to be trained.
- Each one also has its own hyperparameters (number of channels, filter size, etc.).

Representing natural language

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?
gravity

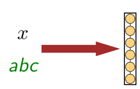
What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?
graupel

Where do water droplets collide with ice crystals to form precipitation?
within a cloud

- Now let us turn our attention to natural language processing. As a motivating example, suppose we wanted to build a question answering system.
- Here are examples of questions from the SQuAD question answering benchmark, where the input is the paragraph, a question, and the goal is to select the span of the paragraph that answers the question.
- This requires somehow relating the paragraph to the question; sometimes string match will work (e.g., "precipitation"), and sometimes they don't (e.g., "causes" and "product").

Embedding tokens

EmbedToken



Looks up the vector for a token.

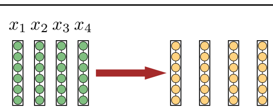
In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

Meaning of words/tokens depends on context...

- Words are discrete objects, whereas neural networks speak vectors.
- So the first step is to embed the words (more generally, tokens) into vectors. This is usually just accomplished by looking up the token in a dictionary that maps tokens to vectors.
- Now given a sentence (sequence of tokens), we just embed each of the tokens independently to produce a sequence of vectors.
- However, this is not quite satisfactory because the meaning of words depends on context. For example, "product" could mean result or it could mean multiplication.

Representing sequences

SequenceModel



Process each element of a sequence with respect to other elements.

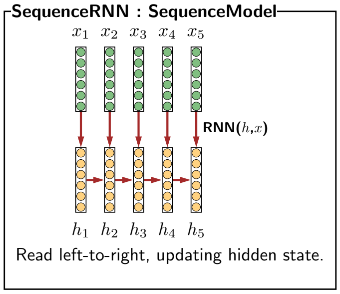
Two implementations:

- Recurrent neural networks
- Transformers

- We're now going to define an abstract function **SequenceModel** (to continue the programming metaphor).
- **SequenceModel** takes a sequence of vectors and produces a corresponding sequence of vectors, where each vector has been "contextualized" with respect to the other vectors.
- We will see two implementations, recurent neural networks and Transformers.

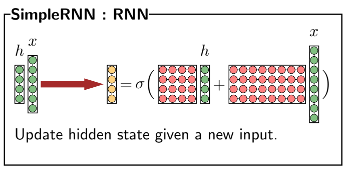
Recurrent neural networks

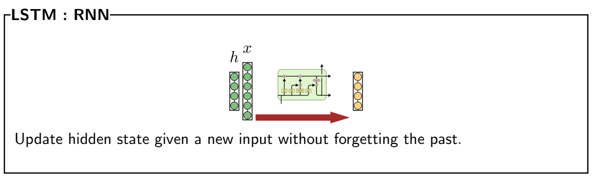
In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.



- A recurrent neural network (RNN) can be thought of as reading left to right. It maintains a hidden state which represents the past and gets updated with each new input vector.
- At the end of the day, however, it still produces a sequence of (contextualized) vectors given a sequence of input vectors.
- We still have to specify how the RNN computes the new hidden state given the old hidden state and the new input.

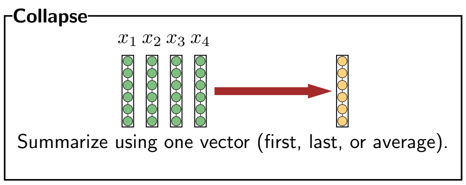
Recurrent neural networks





- There are two types of RNNs I will talk about.
- A Simple RNN takes the old hidden state h, a new input x and produces a new hidden state.
- Both h and x get multiplied by a vector; the result is summed, and we apply the activation function as usual.
- Note that this is equivalent to just applying **FeedForward** on the concatenation of h and x.
- One problem with simple RNNs is that they suffer from the vanishing gradient problem, which makes them hard to train.
- Long Short-Term Memory (LSTMs) were developed in response to this problem. We won't go over the details, but will just say that it privileges h and makes sure that h doesn't forget the history of inputs.

Collapsing to a single vector



Example text classification model:

In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

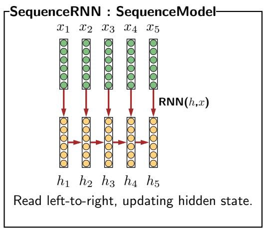
$$\text{score} = \mathbf{w} \cdot \textbf{Collapse}(\textbf{SequenceModel}^3(\textbf{EmbedToken}(\mathrm{x})))$$

- Note that the sequence model produces a variable number of vectors. In order to do classifaction, we need to **Collapse** the vectors into one, which can then be used to drive the score for classification.
- There are three common things that are done: return the first vector, return the last vector, or return the average of all of them.

Long-range dependencies

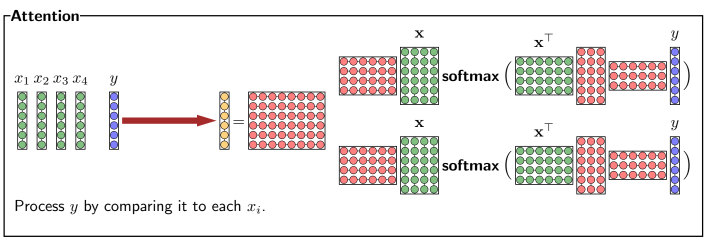
[CLS] What causes precipitation to fall? [SEP] In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

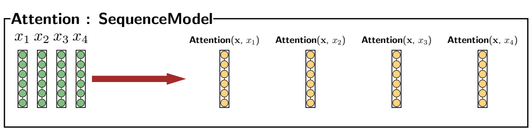
Problem: RNN (and ConvNets) are very local



- This is all nice, but there is one big problem with RNNs, which is that they are very local, meaning that the vector produced at a given position will in practice only depend on a small neighborhood.
- On the other hand, language often has long-range dependencies.
- For the model to leverage this knowledge, it has to remember the first material in the hidden state, update that hidden state repeatedly without forgetting what it learned.
- Instead we would like an architecture that can allow information to propagate more quickly and freely across the sequence. This architecture is the Transformer. But we need to introduce a few preliminaries first.

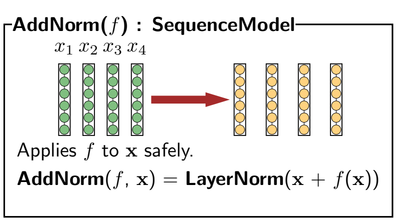
Attention mechanism





- This is a big slide.
- Attention is the core mechanism that allows us to model long-range dependencies effectively.
- Formally, attention takes a collection of input vectors $x_1, \dots, x_n$, a query vector y, and the goal is to process y in the context of the input vectors.
- Let's walk through the diagram slowly. We start with y and reduce its dimensionality. Similary, we can reduce the dimensionality for all rows of $\mathbf{x}^\top$.
- Now the key part is to take the dot product between the representation of y and the representation of each x_i.
- We now have one column representing similarity scores (positive or negative) between y and each x_i.
- We apply the softmax, which exponentiates each component and then normalizes to 1.
- We can use the distribution to take a convex combination of the columns of $\mathbf{x}$. And finally we project onto the desired dimensionality.
- The above calculations is one attention head. In parallel, we go through the same motions to construct another vector. These are concanentated together and projected down to the original space.
- Another form of easy attention (called self-attention) is self-attention, in which case the queries are simply the same elements.
- This provides a sequence model where all pairs of elements of the sequence interact.

Layer normalization and residual connections

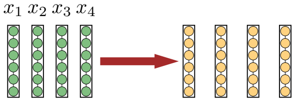


- There are two other pieces we need to introduce before we can fully define the Transformer: layer normalization and residual connections.
- They can be thought of as technical devices to make the final network easier to train.
- Together, they can be packed up into **AddNorm**. The interface is the same as a sequence model: it takes a sequence of vectors and produces a corresponding sequence of vectors.
- **AddNorm** takes a function f and applies it to the input $\mathbf{x}$. Then we add $\mathbf{x}$ (called a residual connection), which allows information from $\mathbf{x}$ to be directly passed through in case f is somehow messed up (say, early in training).
- Then we apply layer normalization, which takes a vector x and makes sure it's not too small or big (or else gradients might vanish or explode). Specifically, it subtracts the average of the elements of x and divides by the standard deviation. We do this for each vector in $\mathbf{x}$.
- In summary, **AddNorm** applies f to $\mathbf{x}$ safely.

Transformer

TransformerBlock : SequenceModel

$x_1\ x_2\ x_3\ x_4$



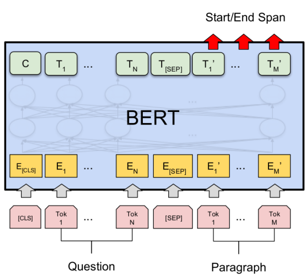
Processes each object x_i in context.

TransformerBlock$(\mathbf{x})$ = **AddNorm**(**FeedForward**, **AddNorm**(**Attention**, $\mathbf{x}$))

- Finally, we are ready to introduce the Transformer, which was introduced in 2017 and has really overthrown RNNs as the sequence model of choice.
- The **TransformerBlock** is a sequence model that takes a sequence of vectors to corresponding sequence of contextual vectors.
- We've already defined all the pieces, so the Transformer is just a one-liner.
- First, we apply self-attention to $\mathbf{x}$ to contextualize the vectors. Then we apply **AddNorm** (layer normalization with residual connections) to make things safe.
- Second, we apply a feedforward network to further process each vector independently. Then we do another **AddNorm**, and that's it.
- Note: in the actual Transformer, the **FeedForward** function has an extra matrix at the end.

[Devlin+, 2018]

BERT



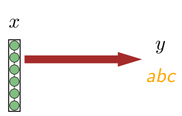
[CLS] What causes precipitation to fall? [SEP] In meterology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity.

BERT$(\mathbf{x})$ = **TransformerBlock**^{24}(**EmbedToken**$(\mathbf{x})$)

- Now we can explain BERT, the large unsupervised pretrained model which transformed NLP in 2018. Before then, there were many specialized architectures for different tasks but BERT was a single model architecture that worked well across many tasks.
- BERT is a function that takes a sentence, turns it into a sequence of vectors, and then just applies a Transformer block 24 times.
- When it's used for question answering, you concatenate the question with the paragraph.
- There are some details omitted: BERT defines tokens as word pieces as opposed to words. It also uses positional encodings as the Transformer block is invariant to the order of the vectors.
- Note that we are also not talking about the objective function (masked language modeling, next sentence prediction) used to train BERT.

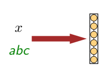
Generating tokens

GenerateToken



Generate token y based on $x\cdot$ **EmbedToken**(y).

EmbedToken



Looks up the vector for a token.

- So far, we've mostly studied how to design functions that can process a sentence (sequence of tokens or vectors).
- We can also generate new sequences.
- The basic building block for generation is something that takes a vector and outputs a token.
- This process is the reverse of **EmbedToken**, but uses it as follows: we compute a score $x\cdot$ **EmbedToken**(y) for each candidate token.
- Then we apply softmax (exponentiate and normalize) to get a distribution over words y. From this distribution, we can either take the token with the highest probability or simply sample a word from this distribution.

Generating sequences

LanguageModel

$\mathbf{x}$

the quick brown → *fox*

Generate next token in the sequence.

LanguageModel$(\mathbf{x})$ = **GenerateToken**(**Collapse**(**SequenceModel**(**EmbedToken**$(\mathbf{x})$)))

- In language modeling, we wish to generate (predict) the next token given the previous words generated so far.
- In this case, we simply take the history, which is a sequence of tokens, embed them, apply a sequence model (e.g., RNN or a Transformer).
- We then **Collapse** this into one vector (usually the last one).
- Now we've reduced the problem to something that **GenerateToken** can solve.

Sequence-to-sequence models

SequenceToSequence

$\mathbf{x}$

la maison bleue —**LanguageModel**→ *the blue house*

Generate a sequence from another sequence.

Applications:

- Machine translation: sentence to translation
- Document summarization: document to summary
- Semantic parsing: sentence to code

- Perhaps the most versatile interface is sequence-to-sequence models, where the model takes as input a sequence of tokens and produces an sequence of tokens.
- Importantly, the input sequence and the output sequence are not in 1:1 correspondence, and in general they have different lengths.
- Sequence-to-sequence models can be reduced to language modeling, where you feed in the input $\mathbf{x}$ along with the output tokens that you've generated so far, and ask for the next token.
- There are a number of applications (mostly in NLP) that uses sequence-to-sequence models: machine translation, document summarization, and semantic parsing to name a few.

Summary

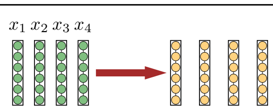
FeedForward **Conv** **MaxPool**

EmbedToken **SequenceRNN** **SimpleRNN** **LSTM**

Attention **AddNorm** **TransformerBlock** **BERT**

Collapse **GenerateToken** **LanguageModel** **SequenceToSequence**

SequenceModel



Process each element of a sequence with respect to other elements.

- In summary, we've done a whirlwind tour over different types of differentiable programs from deep learning.
- We started with feedforward networks (**FeedForward**), and then talked about convolutional neural networks (**Conv**, **MaxPool**).
- Then we considered token sequences (e.g., text), where we always start by embedding the tokens into vectors (**EmbedToken**).
- There are two paths: The first uses RNNs and the second uses Transformers.
- There are many details that we have glossed over, but the thing you should take away is to understand rigorously what the type signature of all these functions are, and some intuition behind what the function is meant to be doing.