Deep Learning for Poets (Part IV)

Amir H. Payberah
payberah@kth.se
20/12/2018

TensorFlow

Linear and Logistic regression

Deep Feedforward Networks

CNN, RNN, Autoencoders

TensorFlow

Linear and Logistic regression

Deep Feedforward Networks

CNN, RNN, Autoencoders

CNN

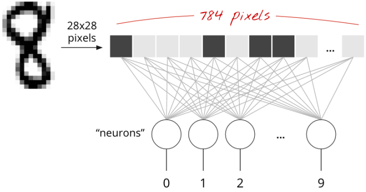
Let's Start With An Example

MNIST Dataset

- Handwritten digits in the MNIST dataset are 28x28 pixel greyscale images.

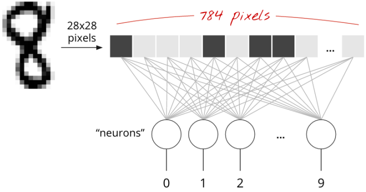
One-Layer Network For Classifying MNIST (1/4)

- Let's make a one-layer neural network for classifying digits.

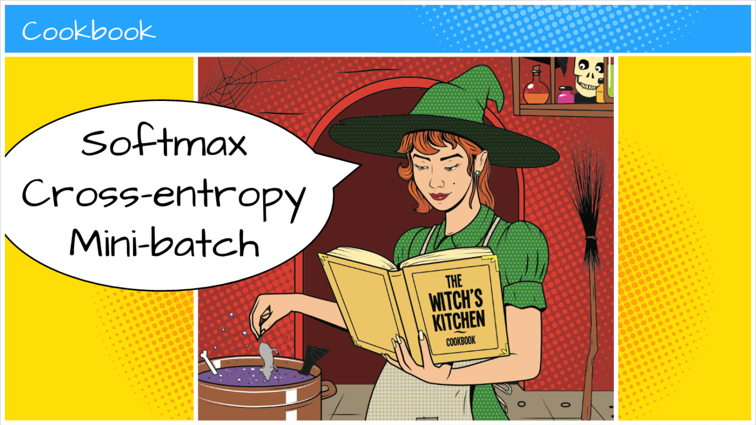


One-Layer Network For Classifying MNIST (1/4)

- Let's make a one-layer neural network for classifying digits.
- Each neuron in a neural network:
  - Does a weighted sum of all of its inputs
  - Adds a bias
  - Feeds the result through some non-linear activation function, e.g., softmax.



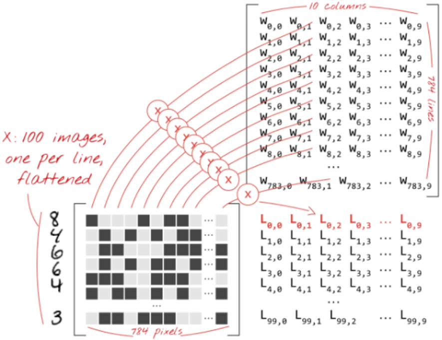
One-Layer Network For Classifying MNIST (2/4)



[https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd]

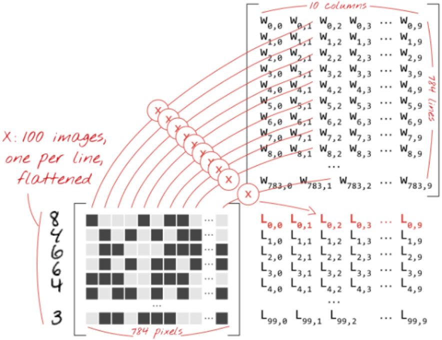
One-Layer Network For Classifying MNIST (3/4)

- Assume we have a batch of 100 images as the input.



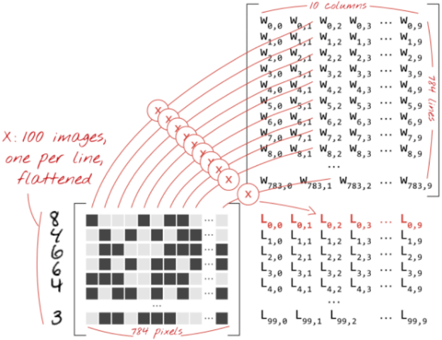
One-Layer Network For Classifying MNIST (3/4)

- Assume we have a batch of 100 images as the input.
- Using the first column of the weights matrix **W**, we compute the weighted sum of all the pixels of the first image.



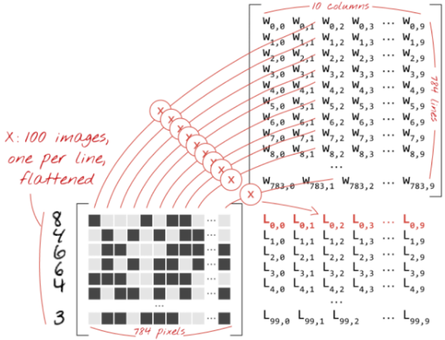
One-Layer Network For Classifying MNIST (3/4)

- Assume we have a batch of 100 images as the input.
- Using the first column of the weights matrix **W**, we compute the weighted sum of all the pixels of the first image.
  - The first neuron:
    $L_{0,0} = w_{0,0}x_0^{(1)} + w_{1,0}x_1^{(1)} + \cdots + w_{783,0}x_{783}^{(1)}$



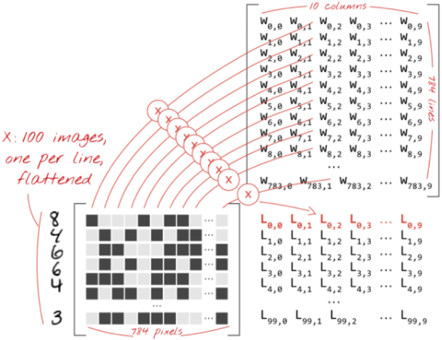
One-Layer Network For Classifying MNIST (3/4)

- Assume we have a batch of 100 images as the input.
- Using the first column of the weights matrix **W**, we compute the weighted sum of all the pixels of the first image.
  - The first neuron:
    $\mathtt{L}_{0,0} = \mathtt{w}_{0,0}\mathtt{x}_0^{(1)} + \mathtt{w}_{1,0}\mathtt{x}_1^{(1)} + \cdots + \mathtt{w}_{783,0}\mathtt{x}_{783}^{(1)}$
  - The 2nd neuron until the 10th:
    $\mathtt{L}_{0,1} = \mathtt{w}_{0,1}\mathtt{x}_0^{(1)} + \mathtt{w}_{1,1}\mathtt{x}_1^{(1)} + \cdots + \mathtt{w}_{783,1}\mathtt{x}_{783}^{(1)}$
    $\cdots$
    $\mathtt{L}_{0,9} = \mathtt{w}_{0,9}\mathtt{x}_0^{(1)} + \mathtt{w}_{1,9}\mathtt{x}_1^{(1)} + \cdots + \mathtt{w}_{783,9}\mathtt{x}_{783}^{(1)}$



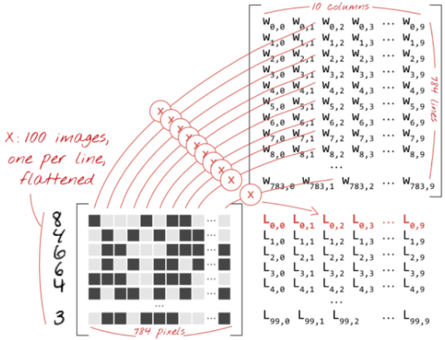
One-Layer Network For Classifying MNIST (3/4)

- Assume we have a batch of 100 images as the input.
- Using the first column of the weights matrix **W**, we compute the weighted sum of all the pixels of the first image.
  - The first neuron:
    $L_{0,0} = w_{0,0}x_0^{(1)} + w_{1,0}x_1^{(1)} + \cdots + w_{783,0}x_{783}^{(1)}$
  - The 2nd neuron until the 10th:
    $L_{0,1} = w_{0,1}x_0^{(1)} + w_{1,1}x_1^{(1)} + \cdots + w_{783,1}x_{783}^{(1)}$
    $\ldots$
    $L_{0,9} = w_{0,9}x_0^{(1)} + w_{1,9}x_1^{(1)} + \cdots + w_{783,9}x_{783}^{(1)}$
  - Repeat the operation for the other 99 images, i.e., $\mathbf{x}^{(2)} \ldots \mathbf{x}^{(100)}$



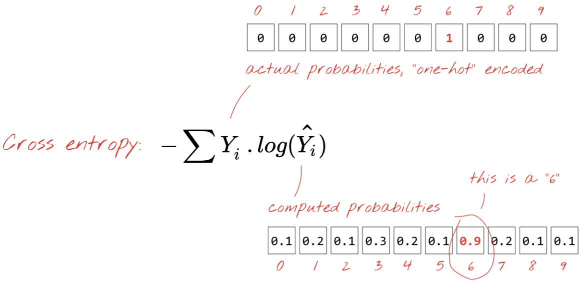
One-Layer Network For Classifying MNIST (4/4)

- Each neuron must now add its bias.
- Apply the softmax activation function for each instance $\mathbf{x}^{(i)}$.
- For each input instance $\mathbf{x}^{(i)}$: $\mathbf{L}_i = \begin{bmatrix} L_{i,0} \\ L_{i,1} \\ \vdots \\ L_{i,9} \end{bmatrix}$
- $\hat{\mathbf{y}}_i = \mathtt{softmax}(\mathbf{L}_i + \mathbf{b})$



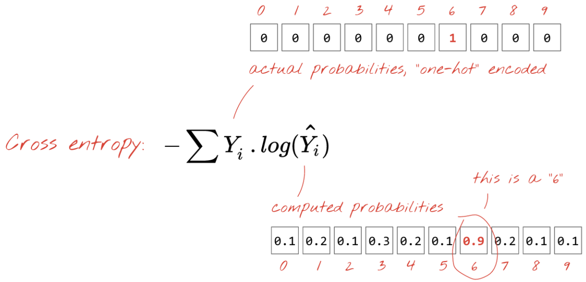
How Good the Predictions Are?

- Define the cost function $J(\mathbf{W})$ as the cross-entropy of what the network tells us ($\hat{\mathbf{y}}_i$) and what we know to be the truth ($\mathbf{y}_i$), for each instance $\mathbf{x}^{(i)}$.



How Good the Predictions Are?

- Define the cost function $\mathtt{J}(\mathbf{W})$ as the cross-entropy of what the network tells us ($\hat{\mathbf{y}}_{\mathtt{i}}$) and what we know to be the truth ($\mathbf{y}_{\mathtt{i}}$), for each instance $\mathbf{x}^{(\mathtt{i})}$.
- Compute the partial derivatives of the cross-entropy with respect to all the weights and all the biases, $\nabla_{\mathbf{W}}\mathtt{J}(\mathbf{W})$.



How Good the Predictions Are?

- Define the cost function $\mathtt{J}(\mathbf{W})$ as the cross-entropy of what the network tells us ($\hat{\mathbf{y}}_{\mathtt{i}}$) and what we know to be the truth ($\mathbf{y}_{\mathtt{i}}$), for each instance $\mathbf{x}^{(\mathtt{i})}$.
- Compute the partial derivatives of the cross-entropy with respect to all the weights and all the biases, $\nabla_{\mathbf{W}}\mathtt{J}(\mathbf{W})$.
- Update weights and biases by a fraction of the gradient $\mathbf{W}^{(\mathtt{next})} = \mathbf{W} - \eta\nabla_{\mathbf{W}}\mathtt{J}(\mathbf{W})$

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	0	0	0

actual probabilities, "one-hot" encoded

Cross entropy: $-\sum Y_i \,.\, log(\hat{Y}_i)$

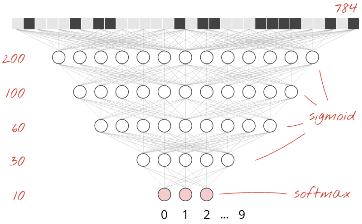
this is a "6"

computed probabilities

0.1	0.2	0.1	0.3	0.2	0.1	0.9	0.2	0.1	0.1
0	1	2	3	4	5	6	7	8	9

Adding More Layers

- Add more layers to improve the accuracy.
- On intermediate layers we will use the the sigmoid activation function.
- We keep softmax as the activation function on the last layer.



[https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd]

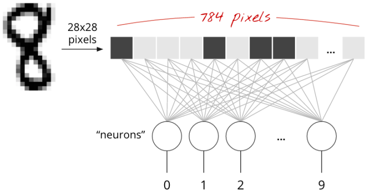
Some Improvement

- Better activation function, e.g., using $\mathtt{ReLU}(z) = \max(0, z)$.
- Overcome Network overfitting, e.g., using dropout.
- Network initialization. e.g., using He initialization.
- Better optimizer, e.g., using Adam optimizer.

[https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd]

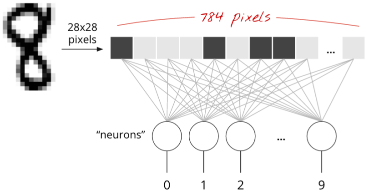
Vanilla Deep Neural Networks Challenges (1/2)

- Pixels of each image were flattened into a single vector (really bad idea).



Vanilla Deep Neural Networks Challenges (1/2)

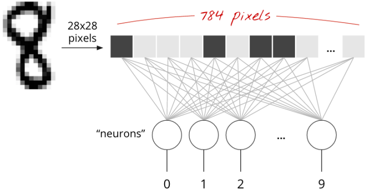
- Pixels of each image were flattened into a single vector (really bad idea).



- Vanilla deep neural networks do not scale.
  - In MNIST, images are black-and-white 28x28 pixel images: $28 \times 28 = 784$ weights.

Vanilla Deep Neural Networks Challenges (1/2)

- Pixels of each image were flattened into a single vector (really bad idea).



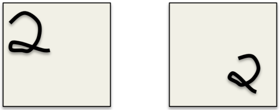
- Vanilla deep neural networks do not scale.
  - In MNIST, images are black-and-white 28x28 pixel images: $28 \times 28 = 784$ weights.
- Handwritten digits are made of shapes and we discarded the shape information when we flattened the pixels.

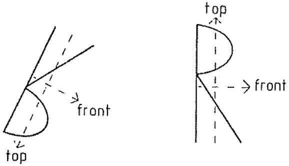
Vanilla Deep Neural Networks Challenges (2/2)

- Difficult to recognize objects.

Vanilla Deep Neural Networks Challenges (2/2)

- Difficult to recognize objects.
- Rotation
- Lighting: objects may look different depending on the level of external lighting.
- Deformation: objects can be deformed in a variety of non-affine ways.
- Scale variation: visual classes often exhibit variation in their size.
- Viewpoint invariance.





Tackle the Challenges

- Convolutional neural networks (CNN) can tackle the vanilla model challenges.
- CNN is a type of neural network that can take advantage of shape information.

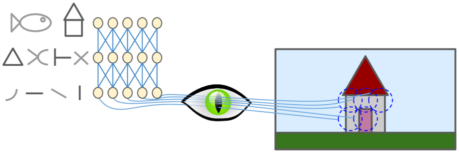
Tackle the Challenges

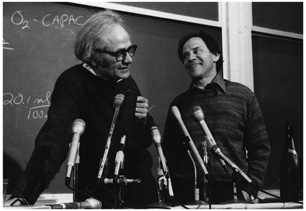
- Convolutional neural networks (CNN) can tackle the vanilla model challenges.
- CNN is a type of neural network that can take advantage of shape information.
- It applies a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification.

Filters and Convolution Operations

Brain Visual Cortex Inspired CNNs

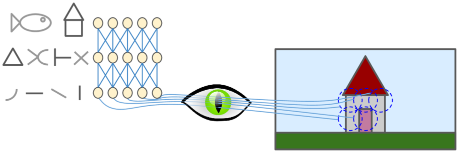
- 1959, David H. Hubel and Torsten Wiesel.
- Many neurons in the visual cortex have a small local receptive field.

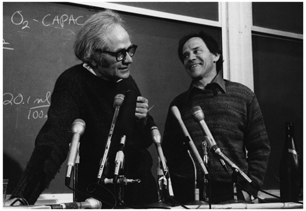




Brain Visual Cortex Inspired CNNs

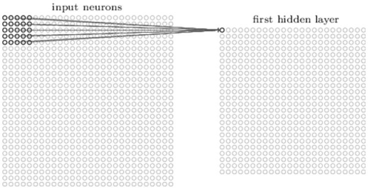
- 1959, David H. Hubel and Torsten Wiesel.
- Many neurons in the visual cortex have a small local receptive field.
- They react only to visual stimuli located in a limited region of the visual field.





Receptive Fields and Filters

- Imagine a flashlight that is shining over the top left of the image.

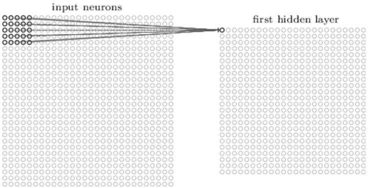


Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Receptive Fields and Filters

- Imagine a flashlight that is shining over the top left of the image.
- The region that it is shining over is called the receptive field.
- This flashlight is called a filter.

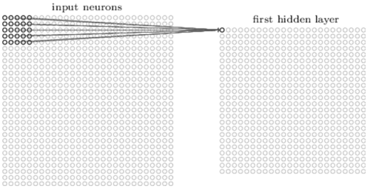


Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Receptive Fields and Filters

- Imagine a flashlight that is shining over the top left of the image.
- The region that it is shining over is called the receptive field.
- This flashlight is called a filter.
- A filter is a set of weights.
- A filter is a feature detector, e.g., straight edges, simple colors, and curves.



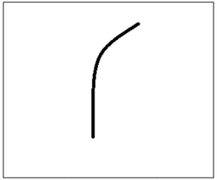
Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Filters Example (1/3)

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

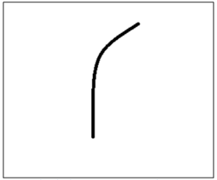


Visualization of a curve detector filter

Filters Example (1/3)

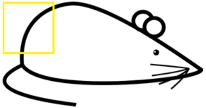
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Original image



Visualization of the filter on the image

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Filters Example (2/3)

Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Filters Example (3/3)

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

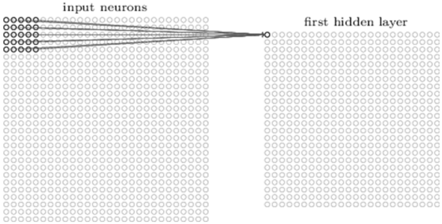
Pixel representation of filter

Multiplication and Summation = 0

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Convolution Operation

- Convolution takes a filter and multiplying it over the entire area of an input image.
- Imagine this flashlight (filter) sliding across all the areas of the input image.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks]

Convolution Operation - More Formal Definition

- Convolution is a mathematical operation on two functions `x` and `h`.
  - You can think of `x` as the input image, and `h` as a filter (kernel) on the input image.

Convolution Operation - More Formal Definition

- Convolution is a mathematical operation on two functions $\mathtt{x}$ and $\mathtt{h}$.
  - You can think of $\mathtt{x}$ as the input image, and $\mathtt{h}$ as a filter (kernel) on the input image.

- For a 1D convolution we can define it as below:

$$\mathtt{y}(\mathtt{k}) = \sum_{\mathtt{n}=0}^{\mathtt{N}-1} \mathtt{h}(\mathtt{n}) \cdot \mathtt{x}(\mathtt{k} - \mathtt{n})$$

- $\mathtt{N}$ is the number of elements in $\mathtt{h}$.

Convolution Operation - More Formal Definition

- Convolution is a mathematical operation on two functions $\mathtt{x}$ and $\mathtt{h}$.
  - You can think of $\mathtt{x}$ as the input image, and $\mathtt{h}$ as a filter (kernel) on the input image.

- For a 1D convolution we can define it as below:

$$\mathtt{y(k)} = \sum_{\mathtt{n=0}}^{\mathtt{N-1}} \mathtt{h(n)} \cdot \mathtt{x(k-n)}$$

- $\mathtt{N}$ is the number of elements in $\mathtt{h}$.

- We are sliding the filter $\mathtt{h}$ over the input image $\mathtt{x}$.

Convolution Operation - 1D Example (1/2)

- Suppose our input 1D image is $\mathtt{x}$, and filter $\mathtt{h}$ are as follows:

$$\mathbf{x} = \boxed{10 \mid 50 \mid 60 \mid 10 \mid 20 \mid 40 \mid 30}$$

$$\mathbf{h} = \boxed{1/3 \mid 1/3 \mid 1/3}$$

- Let's call the output image $\mathtt{y}$.
- What is the value of $\mathtt{y}(3)$?

$$\mathtt{y}(\mathtt{k}) = \sum_{\mathtt{n}=0}^{\mathtt{N}-1} \mathtt{h}(\mathtt{n}) \cdot \mathtt{x}(\mathtt{k}-\mathtt{n})$$

Convolution Operation - 1D Example (2/2)

- To compute $y(3)$, we slide the filter so that it is centered around $x(3)$.

10	50	60	10	20	30	40
0	1/3	1/3	1/3	0	0	0

$$y(3) = \frac{1}{3}50 + \frac{1}{3}60 + \frac{1}{3}10 = 40$$

Convolution Operation - 1D Example (2/2)

- To compute $y(3)$, we slide the filter so that it is centered around $x(3)$.

10	50	60	10	20	30	40
0	1/3	1/3	1/3	0	0	0

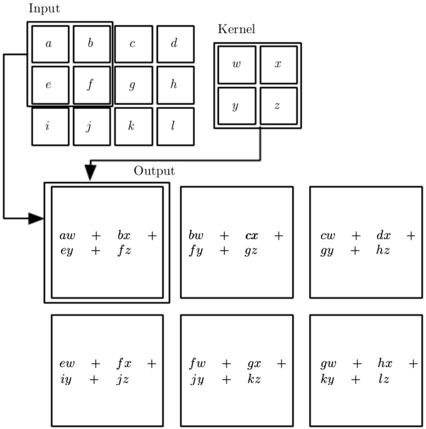
$$y(3) = \frac{1}{3}50 + \frac{1}{3}60 + \frac{1}{3}10 = 40$$

- We can compute the other values of y as well.

$y =$

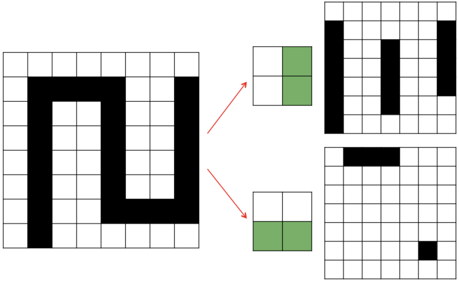
20	40	40	30	20	30	23.333

Convolution Operation - 2D Example (1/2)



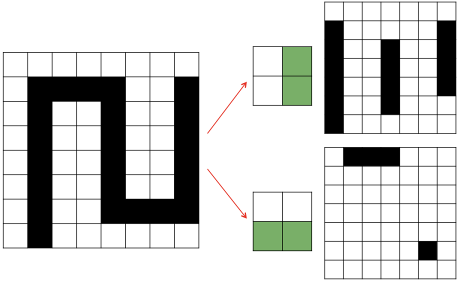
Convolution Operation - 2D Example (2/2)

- Detect vertical and horizontal lines in an image.
- Slide the filters across the entirety of the image.



Convolution Operation - 2D Example (2/2)

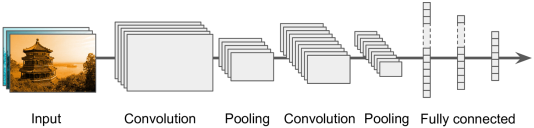
- Detect vertical and horizontal lines in an image.
- Slide the filters across the entirety of the image.
- The result is our feature map: indicates where we've found the feature we're looking for in the original image.



Convolutional Neural Network (CNN)

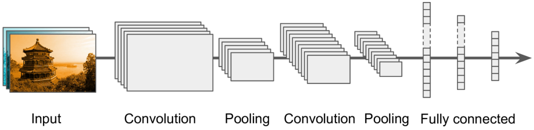
CNN Components

- Convolutional layers: apply a specified number of convolution filters to the image.



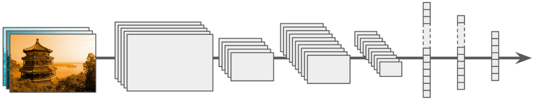
CNN Components

- Convolutional layers: apply a specified number of convolution filters to the image.
- Pooling layers: downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time.

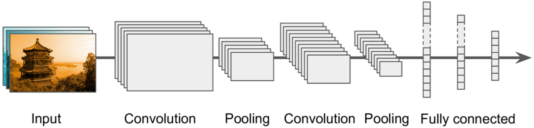


CNN Components

- Convolutional layers: apply a specified number of convolution filters to the image.
- Pooling layers: downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time.
- Dense layers: a fully connected layer that performs classification on the features extracted by the convolutional layers and downsampled by the pooling layers.



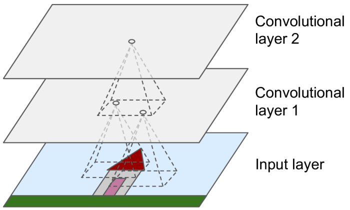
Convolutional Layer



Convolutional Layer (1/3)

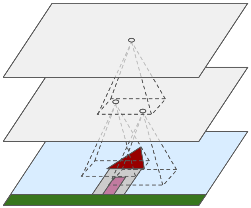
- Sparse interactions

- Each neuron in the convolutional layers is only connected to pixels in its receptive field (not every single pixel).



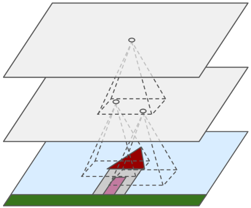
Convolutional Layer (2/3)

- Each neuron applies filters on its receptive field.



Convolutional Layer (2/3)

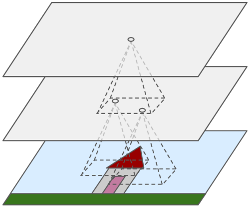
- Each neuron applies filters on its receptive field.
  - Calculates a weighted sum of the input pixels in the receptive field.



Convolutional Layer (2/3)

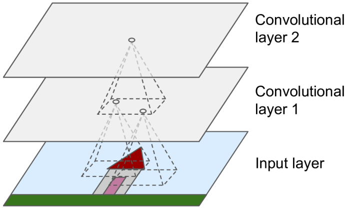
- Each neuron applies filters on its receptive field.
  - Calculates a weighted sum of the input pixels in the receptive field.

- Adds a bias, and feeds the result through its activation function to the next layer.



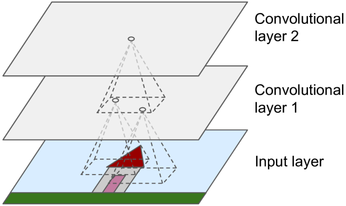
Convolutional Layer (2/3)

- Each neuron applies filters on its receptive field.
  - Calculates a weighted sum of the input pixels in the receptive field.
- Adds a bias, and feeds the result through its activation function to the next layer.
- The output of this layer is a feature map (activation map)



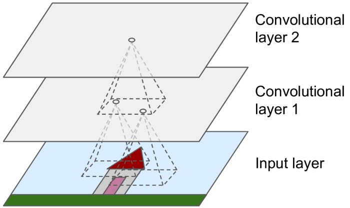
Convolutional Layer (3/3)

- Parameter sharing
- All neurons of a convolutional layer reuse the same weights.



Convolutional Layer (3/3)

- Parameter sharing
- All neurons of a convolutional layer reuse the same weights.
- They apply the same filter in different positions.
- Whereas in a fully-connected network, each neuron had its own set of weights.

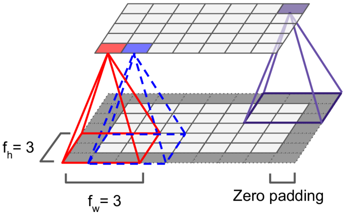


Padding

- What will happen if you apply a 5x5 filter to a 32x32 input volume?
  - The output volume would be 28x28.
  - The spatial dimensions decrease.

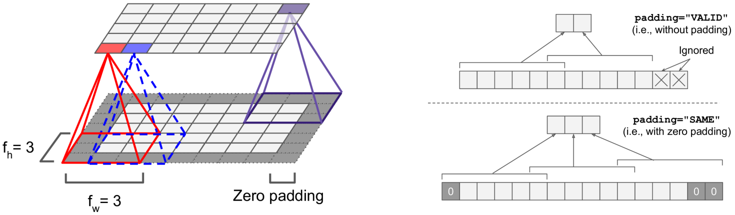
Padding

- What will happen if you apply a 5x5 filter to a 32x32 input volume?
  - The output volume would be 28x28.
  - The spatial dimensions decrease.
- Zero padding: in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.



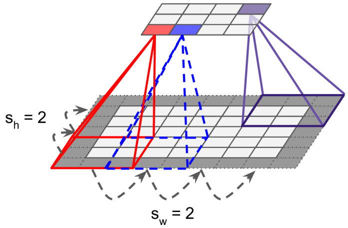
Padding

- What will happen if you apply a 5x5 filter to a 32x32 input volume?
  - The output volume would be 28x28.
  - The spatial dimensions decrease.
- Zero padding: in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.
- In TensorFlow, padding can be either `SAME` or `VALID` to have zero padding or not.



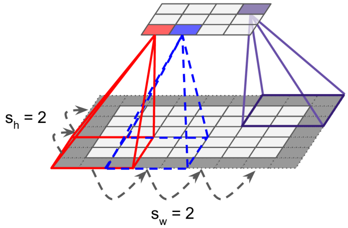
Stride (1/2)

- The distance between two consecutive receptive fields is called the stride.



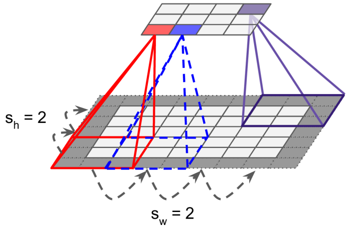
Stride (1/2)

- The distance between two consecutive receptive fields is called the stride.
- The stride controls how the filter convolves around the input volume.

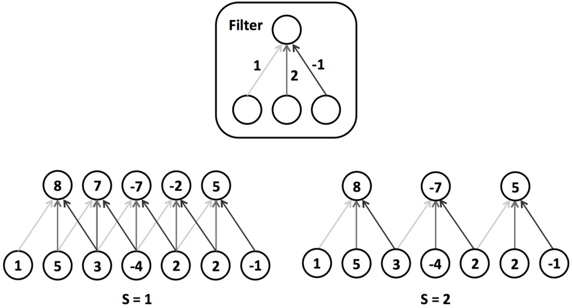


Stride (1/2)

- The distance between two consecutive receptive fields is called the stride.
- The stride controls how the filter convolves around the input volume.
- Assume s_h and s_w are the vertical and horizontal strides, then, a neuron located in row i and column j in a layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, and columns $j \times s_w$ to $j \times s_w + f_w - 1$.

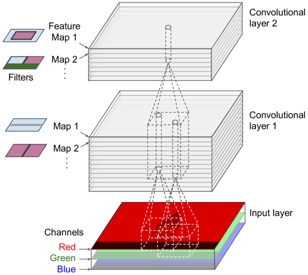


Stride (2/2)



Stacking Multiple Feature Maps

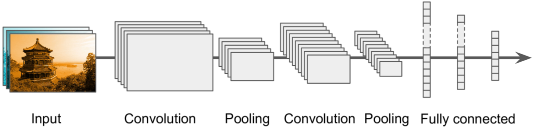
- Up to now, we represented each convolutional layer with a single feature map.
- Each convolutional layer can be composed of several feature maps of equal sizes.
- Input images are also composed of multiple sublayers: one per color channel.
- A convolutional layer simultaneously applies multiple filters to its inputs.



Activation Function

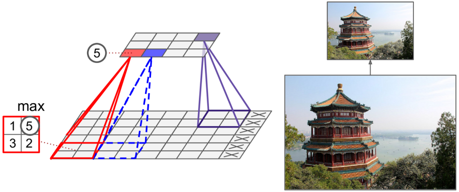
- After calculating a weighted sum of the input pixels in the receptive fields, and adding biases, each neuron feeds the result through its ReLU activation function to the next layer.
- The purpose of this activation function is to add non linearity to the system.

Pooling Layer



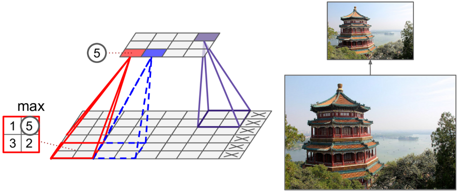
Pooling Layer (1/2)

- After the activation functions, we can apply a pooling layer.
- Its goal is to subsample (shrink) the input image.



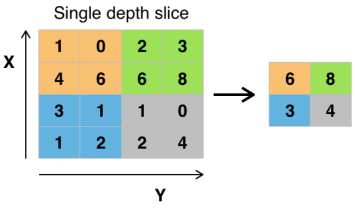
Pooling Layer (1/2)

- After the activation functions, we can apply a pooling layer.
- Its goal is to subsample (shrink) the input image.
  - To reduce the computational load, the memory usage, and the number of parameters.



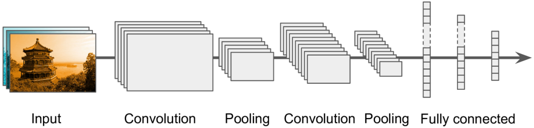
Pooling Layer (2/2)

- Each neuron in a pooling layer is connected to the outputs of a receptive field in the previous layer.
- A pooling neuron has no weights.
- It aggregates the inputs using an aggregation function such as the max or mean.



Example of Maxpool with a 2x2 filter and a stride of 2

Fully Connected Layer



Fully Connected Layer

- This layer takes an input from the last convolution module, and outputs an `N` dimensional vector.
  - `N` is the number of classes that the model has to choose from.

Fully Connected Layer

- This layer takes an input from the last convolution module, and outputs an `N` dimensional vector.
  - `N` is the number of classes that the model has to choose from.
- For example, if you wanted a digit classification model, `N` would be 10.

Fully Connected Layer

- This layer takes an input from the last convolution module, and outputs an `N` dimensional vector.
  - `N` is the number of classes that the model has to choose from.
- For example, if you wanted a digit classification model, `N` would be 10.
- Each number in this `N` dimensional vector represents the probability of a certain class.

Flattening

- We need to convert the output of the convolutional part of the CNN into a 1D feature vector.
- This operation is called flattening.

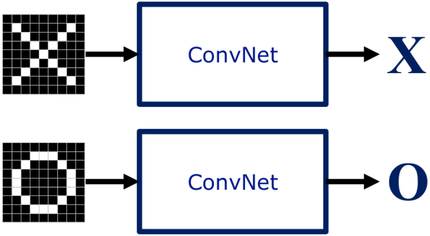
Flattening

- We need to convert the output of the convolutional part of the CNN into a 1D feature vector.
- This operation is called flattening.
- It gets the output of the convolutional layers, flattens all its structure to create a single long feature vector to be used by the dense layer for the final classification.

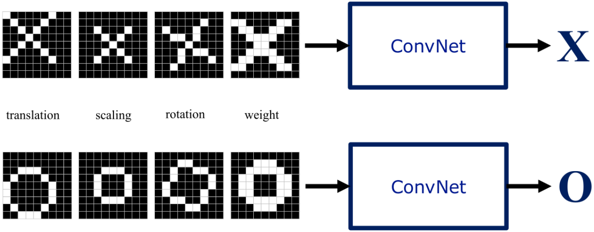
Example

A Toy ConvNet: X's and O's

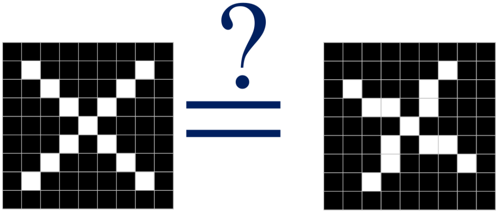
For Example



Trickier Cases



Deciding is Hard



What Computers See

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

$$\stackrel{?}{=}$$

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	1	-1	-1
-1	-1	-1	1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

What Computers See

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

$\stackrel{?}{=}$

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	1	-1	-1
-1	-1	-1	1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	X	-1	-1	-1	-1	X	X	-1
-1	X	X	-1	-1	X	X	-1	-1
-1	-1	X	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	X	-1	-1
-1	-1	X	X	-1	-1	X	X	-1
-1	X	X	-1	-1	-1	-1	X	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

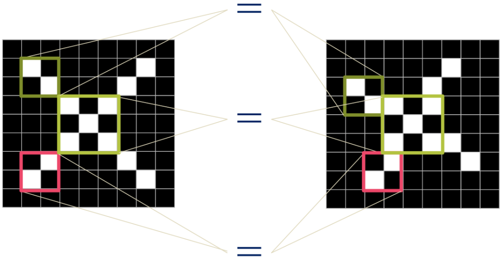
Computers are Literal

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

$\neq$

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	1	-1	-1	-1
-1	-1	1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	1	-1	-1
-1	-1	-1	1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

ConvNets Match Pieces of the Image



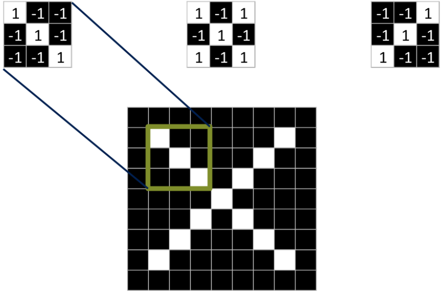
Filters Match Pieces of the Image

1	-1	-1
-1	1	-1
-1	-1	1

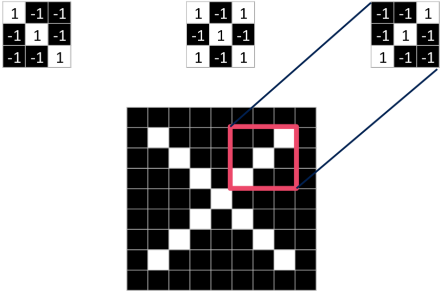
1	-1	1
-1	1	-1
1	-1	1

-1	-1	1
-1	1	-1
1	-1	-1

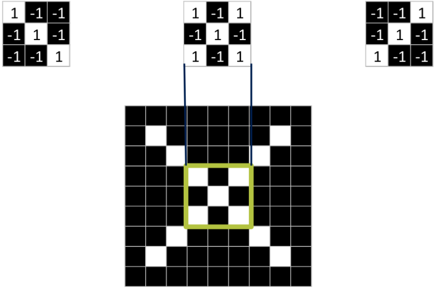
Filters Match Pieces of the Image



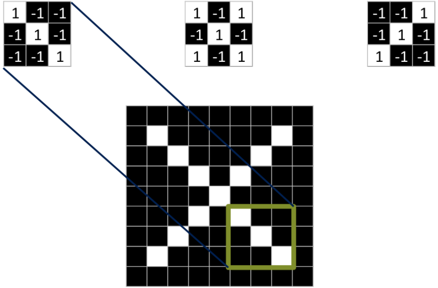
Filters Match Pieces of the Image



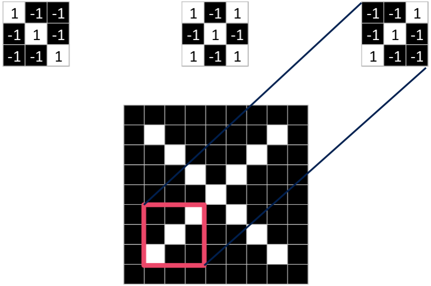
Filters Match Pieces of the Image



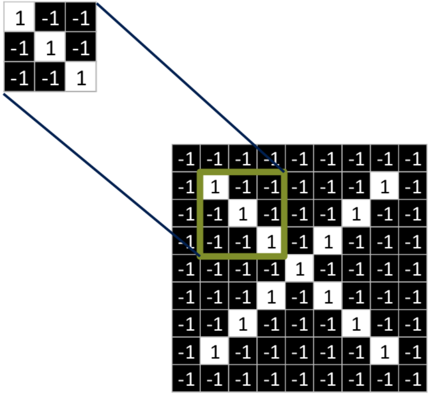
Filters Match Pieces of the Image



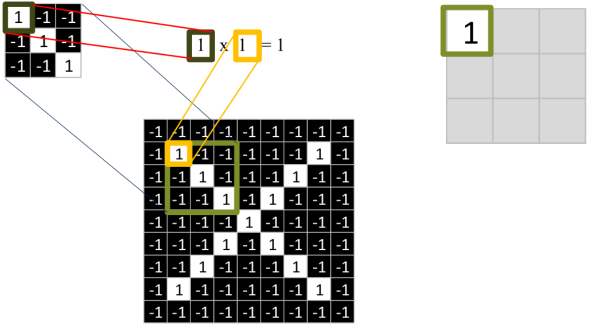
Filters Match Pieces of the Image



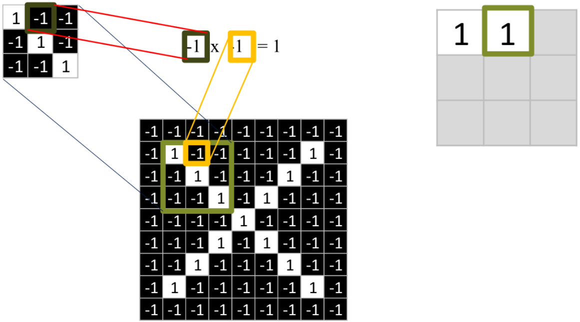
Filtering: The Math Behind the Match



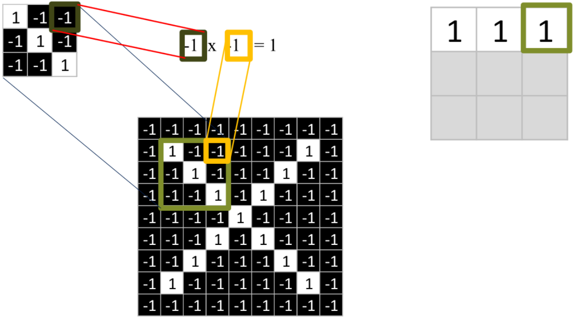
Filtering: The Math Behind the Match



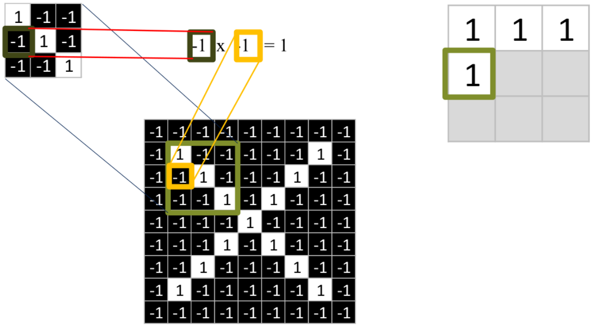
Filtering: The Math Behind the Match



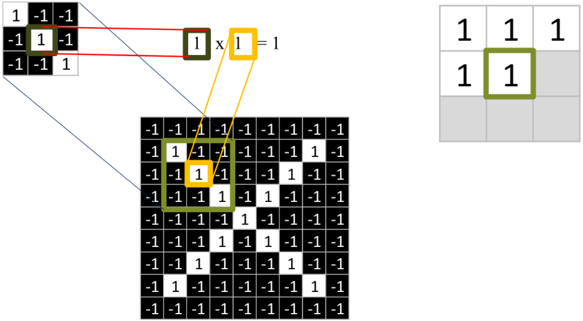
Filtering: The Math Behind the Match

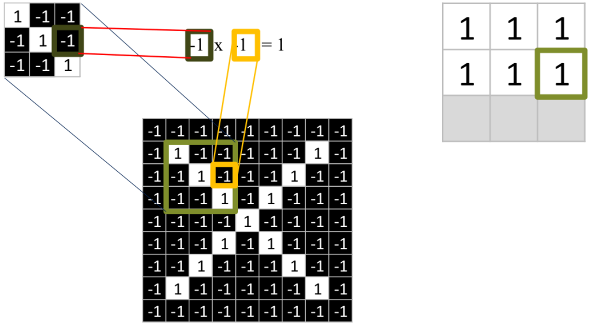


Filtering: The Math Behind the Match

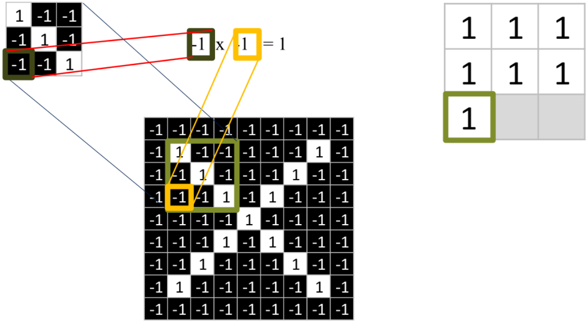


Filtering: The Math Behind the Match

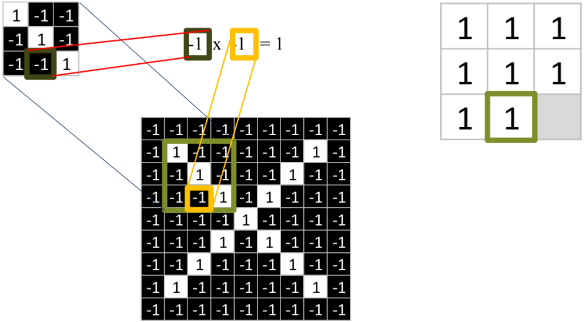



1 -1 -1
-1 1 -1
-1 -1 1
-1 x -1 = 1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 -1 -1 1 -1 -1 -1 -1
-1 -1 -1 1 -1 1 -1 -1 -1
-1 -1 1 -1 -1 -1 1 -1 -1
-1 1 -1 -1 -1 -1 -1 1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
1 1 1
1 1 1

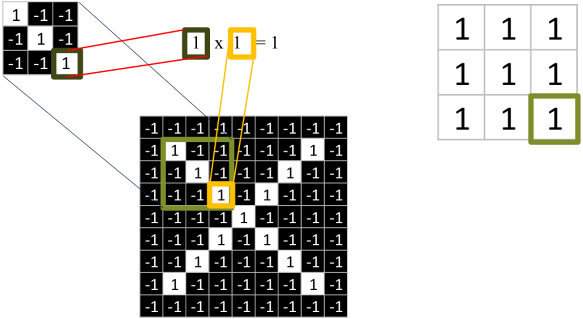
Filtering: The Math Behind the Match



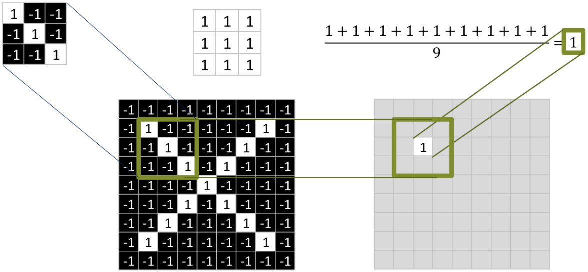
Filtering: The Math Behind the Match



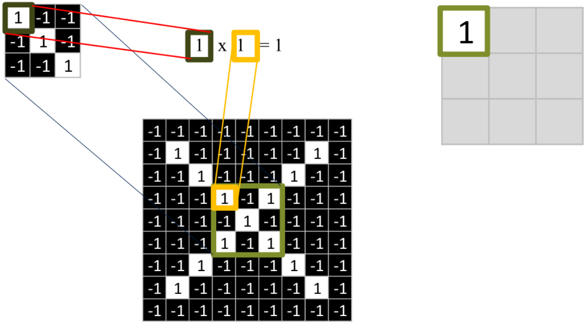
Filtering: The Math Behind the Match



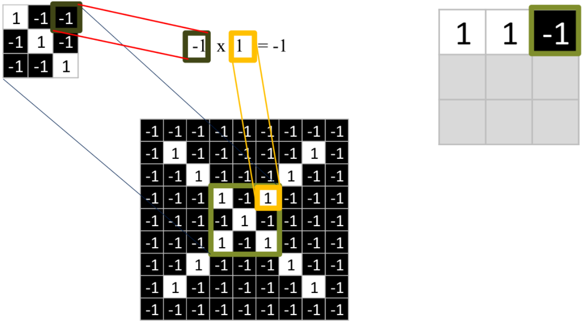
Filtering: The Math Behind the Match



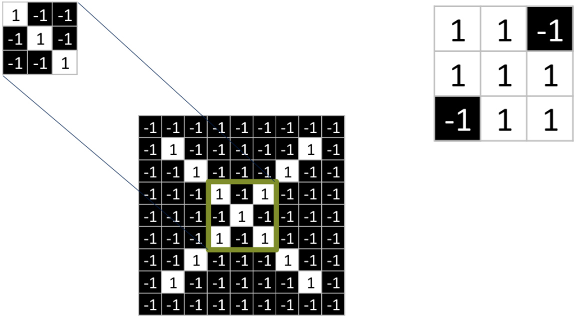
Filtering: The Math Behind the Match



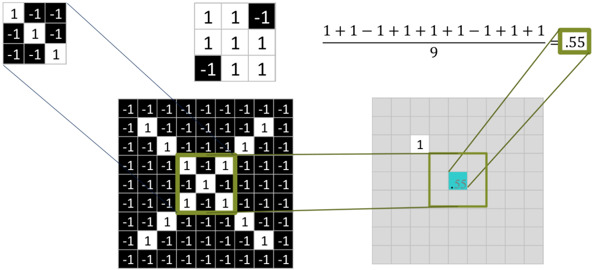
Filtering: The Math Behind the Match



Filtering: The Math Behind the Match



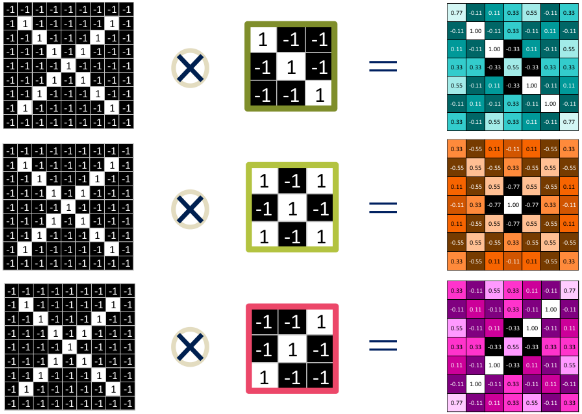
Filtering: The Math Behind the Match



Convolution: Trying Every Possible Match

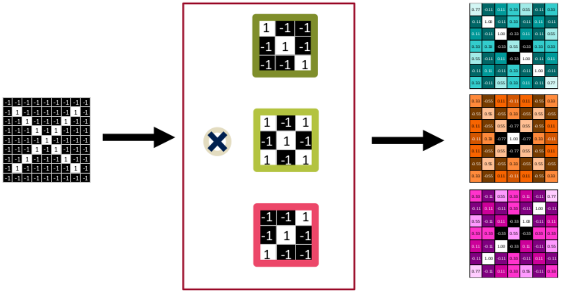
$$
\begin{bmatrix}
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
-1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 \\
-1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 \\
-1 & -1 & -1 & 1 & -1 & 1 & -1 & -1 & -1 \\
-1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 \\
-1 & -1 & -1 & 1 & -1 & 1 & -1 & -1 & -1 \\
-1 & -1 & 1 & -1 & -1 & -1 & 1 & -1 & -1 \\
-1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 \\
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1
\end{bmatrix}
\otimes
\begin{bmatrix}
1 & -1 & -1 \\
-1 & 1 & -1 \\
-1 & -1 & 1
\end{bmatrix}
=
\begin{bmatrix}
0.77 & -0.11 & 0.11 & 0.33 & 0.55 & -0.11 & 0.33 \\
-0.11 & 1.00 & -0.11 & 0.33 & -0.11 & 0.11 & -0.11 \\
0.11 & -0.11 & 1.00 & -0.33 & 0.11 & -0.11 & 0.55 \\
0.33 & 0.33 & -0.33 & 0.55 & -0.33 & 0.33 & 0.33 \\
0.55 & -0.11 & 0.11 & -0.33 & 1.00 & -0.11 & 0.11 \\
-0.11 & 0.11 & -0.11 & 0.33 & -0.11 & 1.00 & -0.11 \\
0.33 & -0.11 & 0.55 & 0.33 & 0.11 & -0.11 & 0.77
\end{bmatrix}
$$

Three Filters Here, So Three Images Out

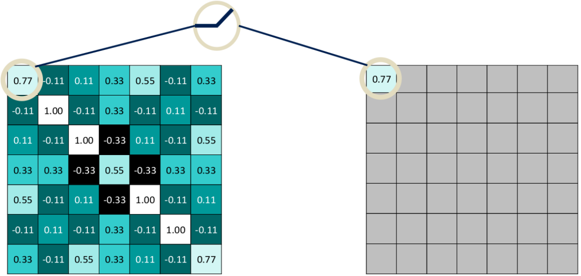


Convolution Layer

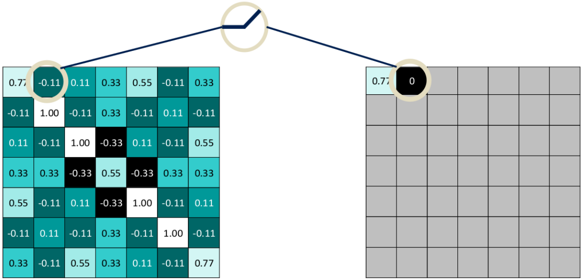
- One image becomes a stack of filtered images.



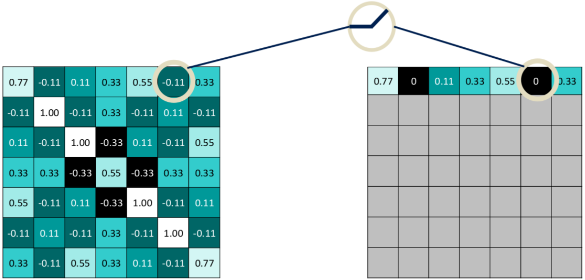
Rectified Linear Units (ReLUs)



Rectified Linear Units (ReLUs)



Rectified Linear Units (ReLUs)



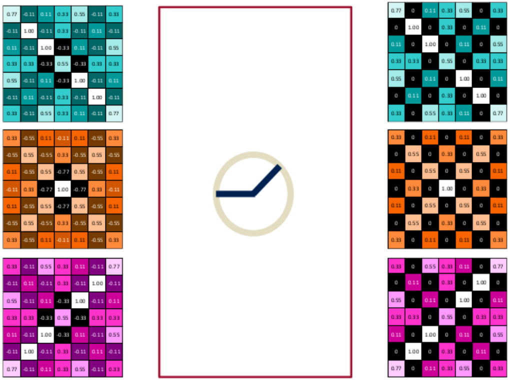
Rectified Linear Units (ReLUs)

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

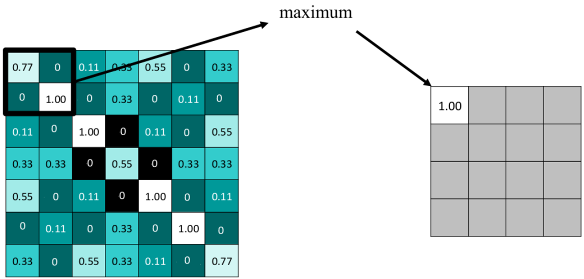
0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	0.77

ReLU Layer

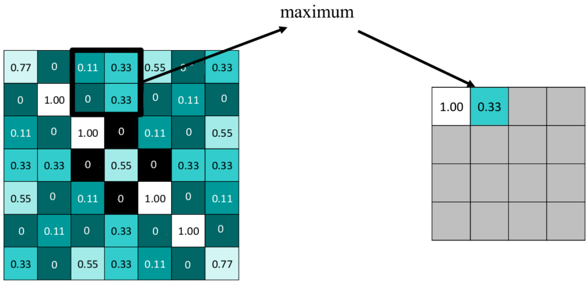
- A stack of images becomes a stack of images with no negative values.



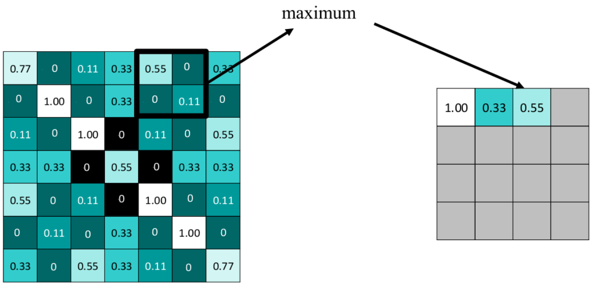
Pooling: Shrinking the Image Stack



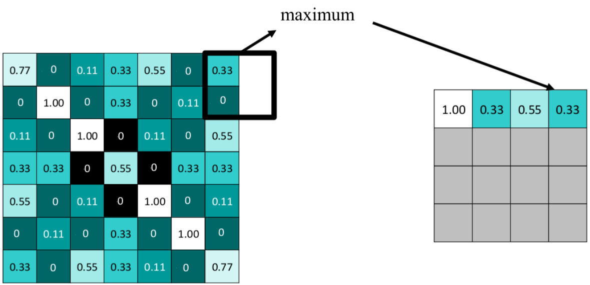
Pooling: Shrinking the Image Stack



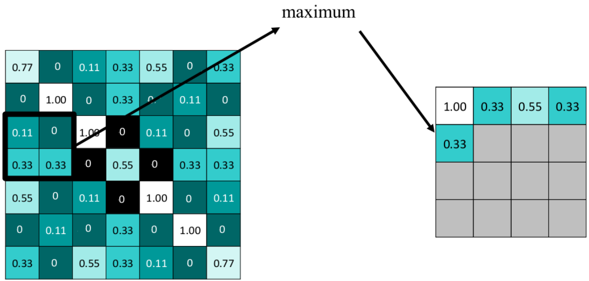
Pooling: Shrinking the Image Stack



Pooling: Shrinking the Image Stack



Pooling: Shrinking the Image Stack



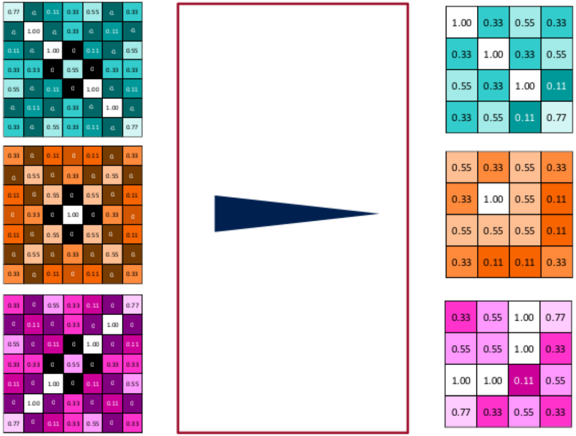
Pooling: Shrinking the Image Stack

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	0.77

max pooling

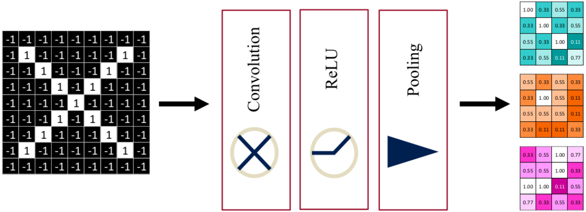
1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

Repeat For All the Filtered Images

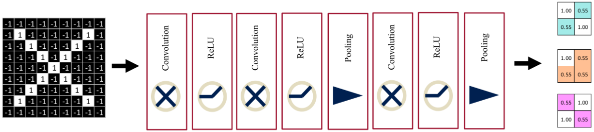


Layers Get Stacked

- The output of one becomes the input of the next.

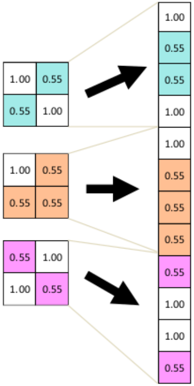


Deep Stacking



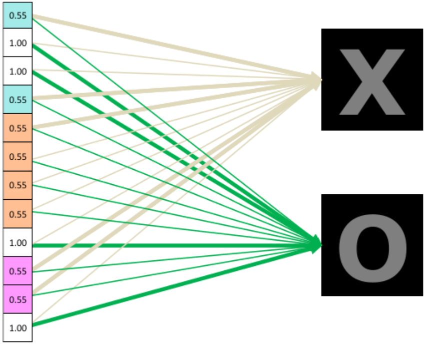
Fully Connected Layer

- Flattening the outputs before giving them to the fully connected layer.

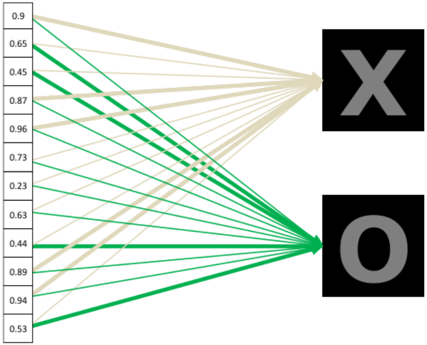


Fully Connected Layer

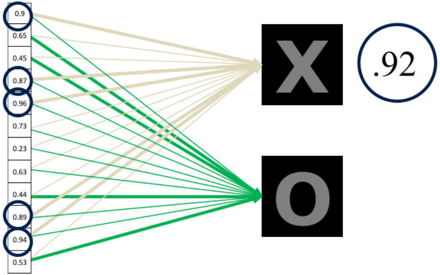
Fully Connected Layer



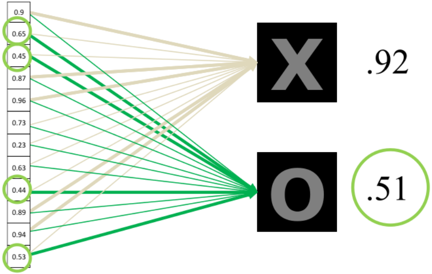
Fully Connected Layer



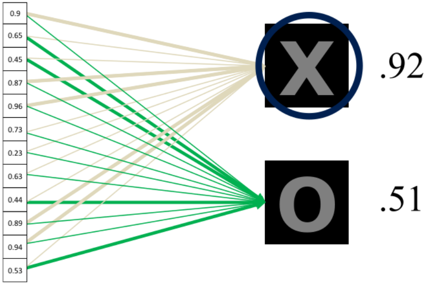
Fully Connected Layer



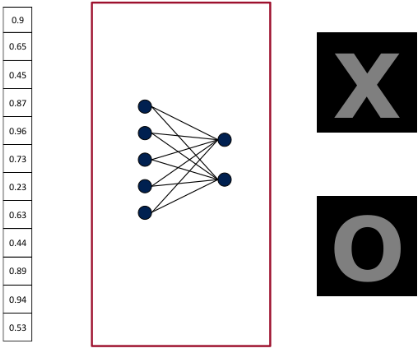
Fully Connected Layer



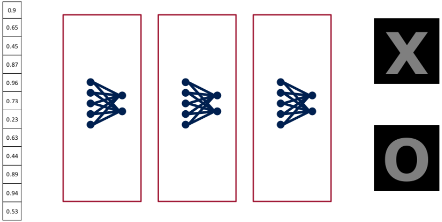
Fully Connected Layer



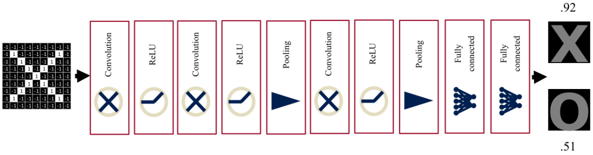
Fully Connected Layer



Fully Connected Layer



Putting It All Together




ENOUGH BLABLA!!
JUST SHOW ME THE CODE!!
imgflip.com

CNN in TensorFlow

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- Conv. layer 2: computes 64 feature maps using a 5x5 filter.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.
- Dense layer: densely connected layer with 1024 neurons.

CNN in TensorFlow (1/8)

- A CNN for the MNIST dataset with the following network.
- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.
- Dense layer: densely connected layer with 1024 neurons.
- Logits layer

CNN in TensorFlow (2/8)

- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.

CNN in TensorFlow (2/8)

- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Input tensor shape: $[\mathtt{batch\_size}, 28, 28, 1]$
- Output tensor shape: $[\mathtt{batch\_size}, 28, 28, 32]$

CNN in TensorFlow (2/8)

- Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- Input tensor shape: [batch_size, 28, 28, 1]
- Output tensor shape: [batch_size, 28, 28, 32]
- Padding same is added to preserve width and height.

```
# MNIST images are 28x28 pixels, and have one color channel
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
y_true = tf.placeholder(tf.float32, [None, 10])

conv1 = tf.layers.conv2d(inputs=X, filters=32, kernel_size=[5, 5], padding="same",
  activation=tf.nn.relu)
```

CNN in TensorFlow (3/8)

- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.

CNN in TensorFlow (3/8)

- Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- Input tensor shape: $[\texttt{batch\_size}, 28, 28, 32]$
- Output tensor shape: $[\texttt{batch\_size}, 14, 14, 32]$

```
pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

CNN in TensorFlow (4/8)

- Conv. layer 2: computes 64 feature maps using a 5x5 filter.

CNN in TensorFlow (4/8)

- Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- Input tensor shape: $[\texttt{batch\_size}, 14, 14, 32]$
- Output tensor shape: $[\texttt{batch\_size}, 14, 14, 64]$

CNN in TensorFlow (4/8)

- Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- Input tensor shape: $[\texttt{batch\_size}, 14, 14, 32]$
- Output tensor shape: $[\texttt{batch\_size}, 14, 14, 64]$
- Padding `same` is added to preserve width and height.

```
conv2 = tf.layers.conv2d(inputs=pool1, filters=64, kernel_size=[5, 5], padding="same",
  activation=tf.nn.relu)
```

CNN in TensorFlow (5/8)

- Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.

CNN in TensorFlow (5/8)

- Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.
- Input tensor shape: $[\texttt{batch\_size}, 14, 14, 64]$
- Output tensor shape: $[\texttt{batch\_size}, 7, 7, 64]$

```
pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
```

CNN in TensorFlow (6/8)

- Flatten tensor into a batch of vectors.

CNN in TensorFlow (6/8)

- Flatten tensor into a batch of vectors.
  - Input tensor shape: $[\mathtt{batch\_size}, 7, 7, 64]$
  - Output tensor shape: $[\mathtt{batch\_size}, 7 * 7 * 64]$

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

CNN in TensorFlow (6/8)

- Flatten tensor into a batch of vectors.
  - Input tensor shape: $[\texttt{batch\_size}, 7, 7, 64]$
  - Output tensor shape: $[\texttt{batch\_size}, 7 * 7 * 64]$

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

- Dense layer: densely connected layer with 1024 neurons.

CNN in TensorFlow (6/8)

- Flatten tensor into a batch of vectors.
  - Input tensor shape: $[\texttt{batch\_size}, 7, 7, 64]$
  - Output tensor shape: $[\texttt{batch\_size}, 7 * 7 * 64]$

```
pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 64])
```

- Dense layer: densely connected layer with 1024 neurons.
  - Input tensor shape: $[\texttt{batch\_size}, 7 * 7 * 64]$
  - Output tensor shape: $[\texttt{batch\_size}, 1024]$

```
dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
```

CNN in TensorFlow (7/8)

- Add dropout operation; 0.6 probability that element will be kept

```
dropout = tf.layers.dropout(inputs=dense, rate=0.4)
```

CNN in TensorFlow (7/8)

- Add dropout operation; 0.6 probability that element will be kept

```
dropout = tf.layers.dropout(inputs=dense, rate=0.4)
```

- Logits layer
  - Input tensor shape: $[\texttt{batch\_size}, 1024]$
  - Output tensor shape: $[\texttt{batch\_size}, 10]$

```
logits = tf.layers.dense(inputs=dropout, units=10)
```

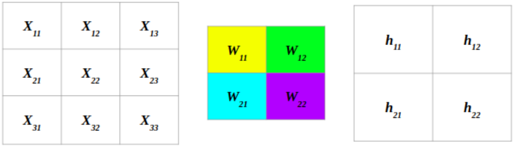
CNN in TensorFlow (8/8)

```
# define the cost and accuracy functions
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=y_true)
cross_entropy = tf.reduce_mean(cross_entropy) * 100

# define the optimizer
lr = 0.003
optimizer = tf.train.AdamOptimizer(lr)
train_step = optimizer.minimize(cross_entropy)

# execute the model
init = tf.global_variables_initializer()

n_epochs = 2000
with tf.Session() as sess:
    sess.run(init)

    for i in range(n_epochs):
        batch_X, batch_y = mnist.train.next_batch(100)
        sess.run(train_step, feed_dict={X: batch_X, y_true: batch_y})
```

Training CNNs

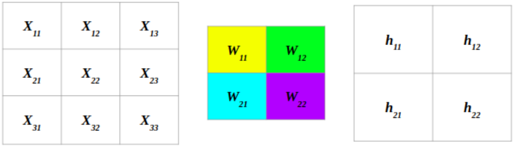
Training CNN (1/4)

- Let's see how to use backpropagation on a single convolutional layer.



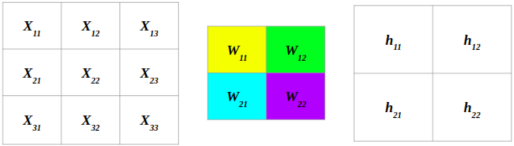
Training CNN (1/4)

- Let's see how to use backpropagation on a single convolutional layer.
- Assume we have an input `X` of size 3x3 and a single filter `W` of size 2x2.



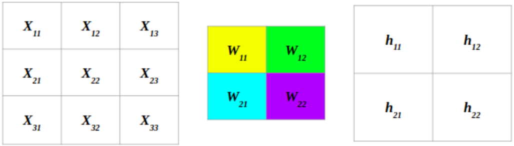
Training CNN (1/4)

- Let's see how to use backpropagation on a single convolutional layer.
- Assume we have an input `X` of size 3x3 and a single filter `W` of size 2x2.
- No padding and stride $= 1$.



Training CNN (1/4)

- Let's see how to use backpropagation on a single convolutional layer.
- Assume we have an input `X` of size 3x3 and a single filter `W` of size 2x2.
- No padding and stride $= 1$.
- It generates an output `H` of size 2x2.

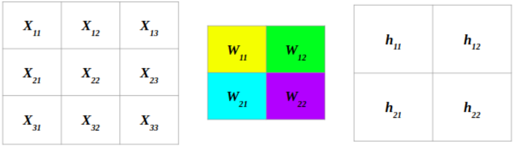


Training CNN (2/4)

- Forward pass

Training CNN (2/4)

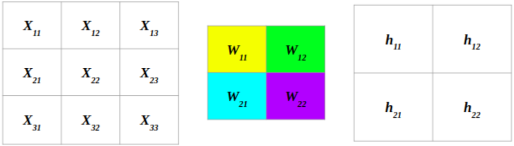
- Forward pass



$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

Training CNN (2/4)

- Forward pass

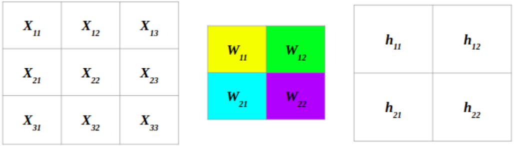


$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

Training CNN (2/4)

- Forward pass



$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

Training CNN (2/4)

- Forward pass

$$
\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}
\quad
\begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}
\quad
\begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}
$$

$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

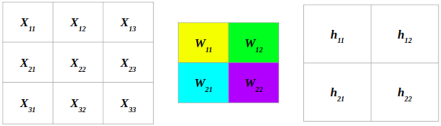
$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

Training CNN (3/4)

- Backward pass
- E is the error: $\mathrm{E} = \mathrm{E}_{h_{11}} + \mathrm{E}_{h_{12}} + \mathrm{E}_{h_{21}} + \mathrm{E}_{h_{22}}$



Training CNN (3/4)

- Backward pass
- E is the error: $\mathtt{E} = \mathtt{E}_{\mathtt{h}_{11}} + \mathtt{E}_{\mathtt{h}_{12}} + \mathtt{E}_{\mathtt{h}_{21}} + \mathtt{E}_{\mathtt{h}_{22}}$

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

W_{11}	W_{12}
W_{21}	W_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$\frac{\partial \mathtt{E}}{\partial \mathtt{W}_{11}} = \frac{\partial \mathtt{E}_{\mathtt{h}_{11}}}{\partial \mathtt{h}_{11}} \frac{\partial \mathtt{h}_{11}}{\partial \mathtt{W}_{11}} + \frac{\partial \mathtt{E}_{\mathtt{h}_{12}}}{\partial \mathtt{h}_{12}} \frac{\partial \mathtt{h}_{12}}{\partial \mathtt{W}_{11}} + \frac{\partial \mathtt{E}_{\mathtt{h}_{21}}}{\partial \mathtt{h}_{21}} \frac{\partial \mathtt{h}_{21}}{\partial \mathtt{W}_{11}} + \frac{\partial \mathtt{E}_{\mathtt{h}_{22}}}{\partial \mathtt{h}_{22}} \frac{\partial \mathtt{h}_{22}}{\partial \mathtt{W}_{11}}$$

Training CNN (3/4)

- Backward pass
- E is the error: $E = E_{h_{11}} + E_{h_{12}} + E_{h_{21}} + E_{h_{22}}$

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

W_{11}	W_{12}
W_{21}	W_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$\frac{\partial E}{\partial W_{11}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial W_{11}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial W_{11}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial W_{11}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial W_{11}}$$

$$\frac{\partial E}{\partial W_{12}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial W_{12}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial W_{12}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial W_{12}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial W_{12}}$$

Training CNN (3/4)

- Backward pass
- E is the error: $\mathrm{E} = \mathrm{E}_{h_{11}} + \mathrm{E}_{h_{12}} + \mathrm{E}_{h_{21}} + \mathrm{E}_{h_{22}}$

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

W_{11}	W_{12}
W_{21}	W_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$
\begin{aligned}
\frac{\partial E}{\partial W_{11}} &= \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{11}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{11}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{11}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{11}} \\
\frac{\partial E}{\partial W_{12}} &= \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{12}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{12}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{12}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{12}} \\
\frac{\partial E}{\partial W_{21}} &= \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{21}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{21}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{21}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{21}}
\end{aligned}
$$

Training CNN (3/4)

- Backward pass
- E is the error: $E = E_{h_{11}} + E_{h_{12}} + E_{h_{21}} + E_{h_{22}}$

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

W_{11}	W_{12}
W_{21}	W_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$\frac{\partial E}{\partial W_{11}} = \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{11}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{11}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{11}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{11}}$$

$$\frac{\partial E}{\partial W_{12}} = \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{12}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{12}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{12}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{12}}$$

$$\frac{\partial E}{\partial W_{21}} = \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{21}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{21}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{21}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{21}}$$

$$\frac{\partial E}{\partial W_{22}} = \frac{\partial E_{h_{11}}}{\partial h_{11}}\frac{\partial h_{11}}{\partial W_{22}} + \frac{\partial E_{h_{12}}}{\partial h_{12}}\frac{\partial h_{12}}{\partial W_{22}} + \frac{\partial E_{h_{21}}}{\partial h_{21}}\frac{\partial h_{21}}{\partial W_{22}} + \frac{\partial E_{h_{22}}}{\partial h_{22}}\frac{\partial h_{22}}{\partial W_{22}}$$

Training CNN (4/4)

- Update the wights W

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

W_{11}	W_{12}
W_{21}	W_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$W_{11}^{(\text{next})} = W_{11} - \eta \frac{\partial E}{\partial W_{11}}$$

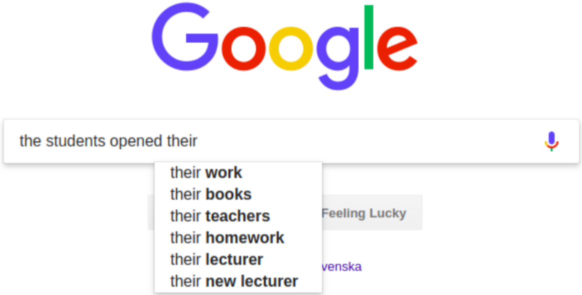
$$W_{12}^{(\text{next})} = W_{12} - \eta \frac{\partial E}{\partial W_{12}}$$

$$W_{21}^{(\text{next})} = W_{21} - \eta \frac{\partial E}{\partial W_{21}}$$

$$W_{22}^{(\text{next})} = W_{22} - \eta \frac{\partial E}{\partial W_{22}}$$

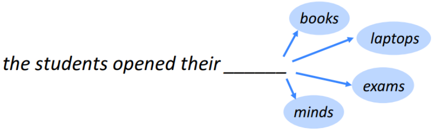
RNN

Let's Start With An Example


Google
the students opened their
their work
their books
their teachers
their homework
their lecturer
their new lecturer
Feeling Lucky
venska

Language Modeling (1/2)

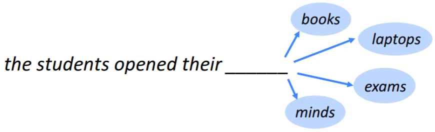
- Language modeling is the task of predicting what word comes next.



Language Modeling (2/2)

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \cdots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$p(x^{(t+1)} = w_j | x^{(t)}, \cdots x^{(1)})$$

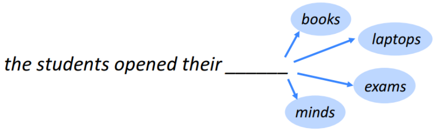


Language Modeling (2/2)

- More formally: given a sequence of words $\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \cdots, \mathtt{x}^{(t)}$, compute the probability distribution of the next word $\mathtt{x}^{(t+1)}$:

$$\mathtt{p}(\mathtt{x}^{(t+1)} = \mathtt{w}_\mathtt{j} | \mathtt{x}^{(t)}, \cdots \mathtt{x}^{(1)})$$

- $\mathtt{w}_\mathtt{j}$ is a word in vocabulary $\mathtt{V} = \{\mathtt{w}_1, \cdots, \mathtt{w}_\mathtt{v}\}$.



n-gram Language Models

- `the students opened their ___`

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
  - Unigrams: `"the", "students", "opened", "their"`

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
  - Unigrams: `"the"`, `"students"`, `"opened"`, `"their"`
  - Bigrams: `"the students"`, `"students opened"`, `"opened their"`

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
  - Unigrams: `"the", "students", "opened", "their"`
  - Bigrams: `"the students", "students opened", "opened their"`
  - Trigrams: `"the students opened", "students opened their"`

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
  - Unigrams: `"the", "students", "opened", "their"`
  - Bigrams: `"the students", "students opened", "opened their"`
  - Trigrams: `"the students opened", "students opened their"`
  - 4-grams: `"the students opened their"`

n-gram Language Models

- `the students opened their ___`
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
  - Unigrams: `"the", "students", "opened", "their"`
  - Bigrams: `"the students", "students opened", "opened their"`
  - Trigrams: `"the students opened", "students opened their"`
  - 4-grams: `"the students opened their"`
- Collect statistics about how frequent different n-grams are, and use these to predict next word.

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
  - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ $\underbrace{\textit{students opened their}}_{\text{condition on this}}$ ______

discard

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
  - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~as the proctor started the clock, the~~ students opened their ______

discard

condition on this

$$p(w_j \mid \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
  - $\mathtt{x}^{(t+1)}$ depends only on the preceding 3 words $\{\mathtt{x}^{(t)}, \mathtt{x}^{(t-1)}, \mathtt{x}^{(t-2)}\}$.

~~*as the proctor started the clock, the*~~ *students opened their* ____

discard

condition on this

$$p(\mathtt{w_j} | \text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

- In the corpus:
  - `"students opened their"` occurred 1000 times

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
  - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

$$\underbrace{\cancel{\textit{as the proctor started the clock, the}}}_{\text{discard}}\ \underbrace{\textit{students opened their}}_{\text{condition on this}}\ \_\_\_\_$$

$$p(w_j|\text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- In the corpus:
  - "students opened their" occurred 1000 times
  - "students opened their books occurred 400 times:
    $p(\text{books}|\text{students opened their}) = 0.4$

n-gram Language Models - Example

- Suppose we are learning a 4-gram Language Model.
  - $x^{(t+1)}$ depends only on the preceding 3 words $\{x^{(t)}, x^{(t-1)}, x^{(t-2)}\}$.

~~*as the proctor started the clock, the*~~ *students opened their* ______

discard

condition on this

$$p(w_j | \text{students opened their}) = \frac{\text{students opened their } w_j}{\text{students opened their}}$$

- In the corpus:
  - "students opened their" occurred 1000 times
  - "students opened their books occurred 400 times:
    $p(\text{books}|\text{students opened their}) = 0.4$
  - "students opened their exams occurred 100 times:
    $p(\text{exams}|\text{students opened their}) = 0.1$

Problems with n-gram Language Models - Sparsity

$$p(\mathtt{w_j} | \text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

Problems with n-gram Language Models - Sparsity

$$p(\mathtt{w_j}|\text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

- What if "`students opened their` $\mathtt{w_j}$" never occurred in data? Then $\mathtt{w_j}$ has probability 0!

Problems with n-gram Language Models - Sparsity

$$p(\mathtt{w_j}|\text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

- What if "`students opened their` $\mathtt{w_j}$" never occurred in data? Then $\mathtt{w_j}$ has probability 0!
- What if "`students opened their`" never occurred in data? Then we can't calculate probability for any $\mathtt{w_j}$!

Problems with n-gram Language Models - Sparsity

$$p(\mathtt{w_j}|\text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

- What if "`students opened their` $\mathtt{w_j}$" never occurred in data? Then $\mathtt{w_j}$ has probability 0!
- What if "`students opened their`" never occurred in data? Then we can't calculate probability for any $\mathtt{w_j}$!
- Increasing `n` makes sparsity problems worse.
  - Typically we can't have `n` bigger than 5.

Problems with n-gram Language Models - Storage

$$p(\mathtt{w_j}|\text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

Problems with n-gram Language Models - Storage

$$p(\mathtt{w_j}|\text{students opened their}) = \frac{\text{students opened their } \mathtt{w_j}}{\text{students opened their}}$$

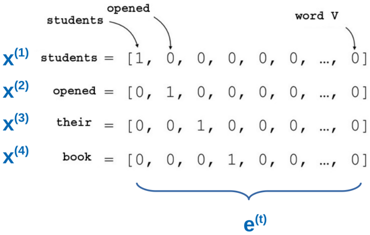
- For "$\mathtt{students\ opened\ their\ w_j}$", we need to store count for all possible 4-grams.
- The model size is in the order of $\mathtt{O(exp(n))}$.
- Increasing $\mathtt{n}$ makes model size huge.

Can We Build a Neural Language Model? (1/3)

- Recall the Language Modeling task:
  - Input: sequence of words $\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \cdots, \mathtt{x}^{(t)}$
  - Output: probability dist of the next word $\mathtt{p}(\mathtt{x}^{(t+1)} = \mathtt{w}_j | \mathtt{x}^{(t)}, \cdots, \mathtt{x}^{(1)})$

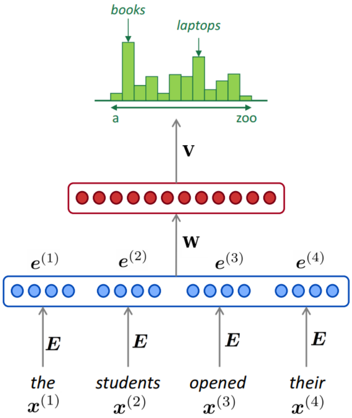
Can We Build a Neural Language Model? (1/3)

- Recall the Language Modeling task:
  - Input: sequence of words $\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \cdots, \mathtt{x}^{(t)}$
  - Output: probability dist of the next word $\mathtt{p}(\mathtt{x}^{(t+1)} = \mathtt{w}_j | \mathtt{x}^{(t)}, \cdots, \mathtt{x}^{(1)})$
- One-Hot encoding
  - Represent a categorical variable as a binary vector.
  - All recodes are zero, except the index of the integer, which is one.
  - Each embedded word $\mathbf{e}^{(t)} = \mathbf{E}^\intercal \mathtt{x}^{(t)}$ is a one-hot vector of size vocabulary size.



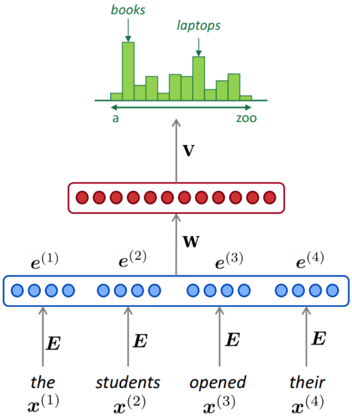
Can We Build a Neural Language Model? (2/3)

- A MLP model
  - Input: words $\mathtt{x}^{(1)}, \mathtt{x}^{(2)}, \mathtt{x}^{(3)}, \mathtt{x}^{(4)}$
  - Input layer: one-hot vectors $\mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \mathbf{e}^{(3)}, \mathbf{e}^{(4)}$
  - Hidden layer: $\mathbf{h} = \mathtt{f}(\mathbf{w}^\intercal \mathbf{e})$, $\mathtt{f}$ is an activation function.
  - Output: $\hat{\mathbf{y}} = \mathtt{softmax}(\mathbf{v}^\intercal \mathbf{h})$



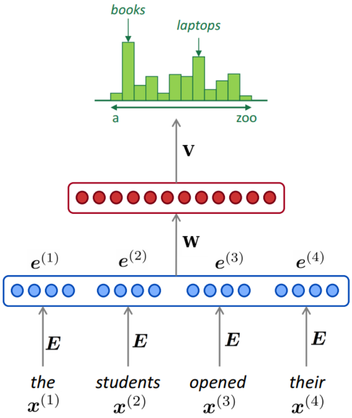
Can We Build a Neural Language Model? (3/3)

- Improvements over n-gram LM:
  - No sparsity problem
  - Model size is `O(n)` not `O(exp(n))`



Can We Build a Neural Language Model? (3/3)

- Improvements over n-gram LM:
  - No sparsity problem
  - Model size is `O(n)` not `O(exp(n))`
- Remaining problems:
  - It is fixed 4 in our example, which is small
  - We need a neural architecture that can process any length input



Recurrent Neural Networks (RNN)

Recurrent Neural Networks (1/4)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.

Recurrent Neural Networks (1/4)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.

Recurrent Neural Networks (1/4)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.
  - It is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).

Recurrent Neural Networks (1/4)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.
  - It is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).

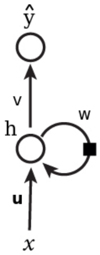
- They can analyze time series data and predict the future.

Recurrent Neural Networks (1/4)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.
  - It is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).
- They can analyze time series data and predict the future.
- They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs.

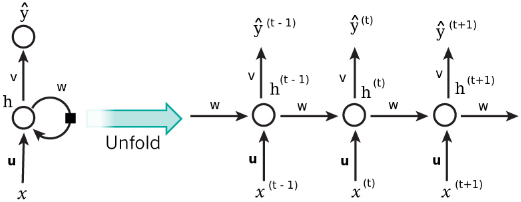
Recurrent Neural Networks (2/4)

- Neurons in an RNN have connections pointing backward.
- RNNs have memory, which captures information about what has been calculated so far.



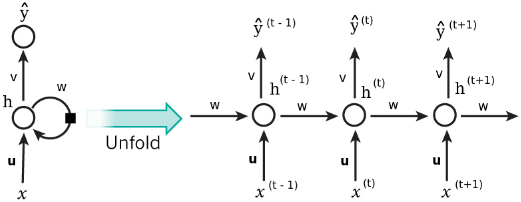
Recurrent Neural Networks (3/4)

- Unfolding the network: represent a network against the time axis.
  - We write out the network for the complete sequence.



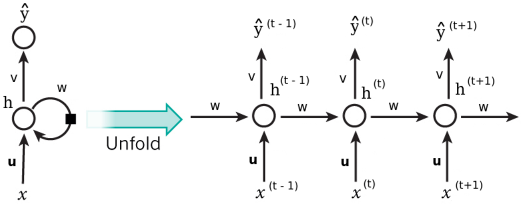
Recurrent Neural Networks (3/4)

- Unfolding the network: represent a network against the time axis.
  - We write out the network for the complete sequence.
- For example, if the sequence we care about is a sentence of three words, the network would be unfolded into a 3-layer neural network.
  - One layer for each word.



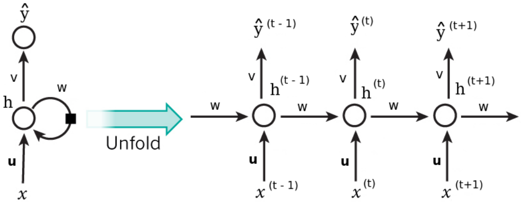
Recurrent Neural Networks (4/4)

- $\mathtt{h}^{(t)} = \mathtt{f}(\mathbf{u}^\intercal \mathbf{x}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)})$, where $\mathtt{f}$ is an activation function, e.g., tanh or ReLU.



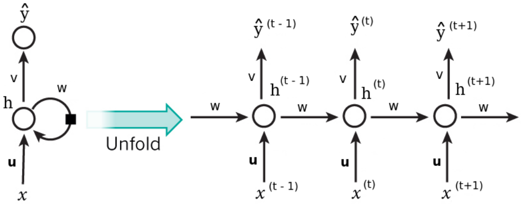
Recurrent Neural Networks (4/4)

- $\mathtt{h}^{(t)} = \mathtt{f}(\mathbf{u}^\intercal \mathbf{x}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)})$, where $\mathtt{f}$ is an activation function, e.g., `tanh` or `ReLU`.
- $\hat{\mathtt{y}}^{(t)} = \mathtt{g}(\mathtt{v}\mathtt{h}^{(t)})$, where $\mathtt{g}$ can be the `softmax` function.



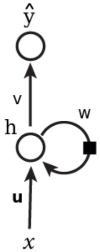
Recurrent Neural Networks (4/4)

- $\mathtt{h}^{(\mathtt{t})} = \mathtt{f}(\mathbf{u}^\intercal \mathbf{x}^{(\mathtt{t})} + \mathtt{w}\mathtt{h}^{(\mathtt{t}-1)})$, where $\mathtt{f}$ is an activation function, e.g., `tanh` or `ReLU`.
- $\hat{\mathtt{y}}^{(\mathtt{t})} = \mathtt{g}(\mathtt{v}\mathtt{h}^{(\mathtt{t})})$, where $\mathtt{g}$ can be the `softmax` function.
- $\mathtt{cost}(\mathtt{y}^{(\mathtt{t})}, \hat{\mathtt{y}}^{(\mathtt{t})}) = \mathtt{cross\_entropy}(\mathtt{y}^{(\mathtt{t})}, \hat{\mathtt{y}}^{(\mathtt{t})}) = -\sum \mathtt{y}^{(\mathtt{t})} \mathtt{log}\hat{\mathtt{y}}^{(\mathtt{t})}$
- $\mathtt{y}^{(\mathtt{t})}$ is the correct word at time step $\mathtt{t}$, and $\hat{\mathtt{y}}^{(\mathtt{t})}$ is the prediction.



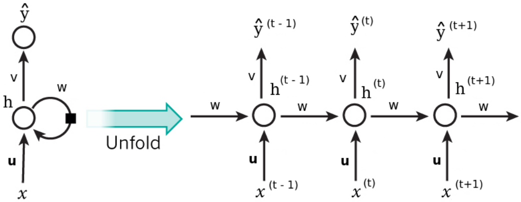
Recurrent Neurons - Weights (1/4)

- Each recurrent neuron has three sets of weights: $\mathbf{u}$, $\mathtt{w}$, and $\mathtt{v}$.



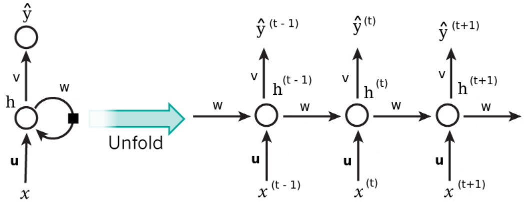
Recurrent Neurons - Weights (2/4)

- **u**: the weights for the inputs $\mathbf{x}^{(t)}$.



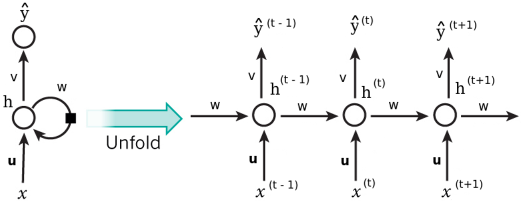
Recurrent Neurons - Weights (2/4)

- **u**: the weights for the inputs $\mathbf{x}^{(t)}$.
- $\mathbf{x}^{(t)}$: is the input at time step t.
- For example, $\mathbf{x}^{(1)}$ could be a one-hot vector corresponding to the first word of a sentence.



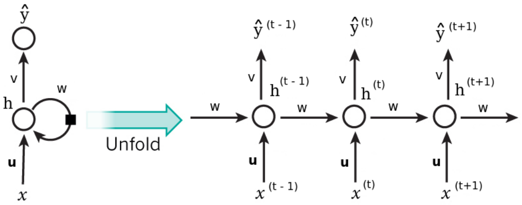
Recurrent Neurons - Weights (3/4)

- $\mathtt{w}$: the weights for the hidden state of the previous time step $\mathtt{h}^{(\mathtt{t}-1)}$.



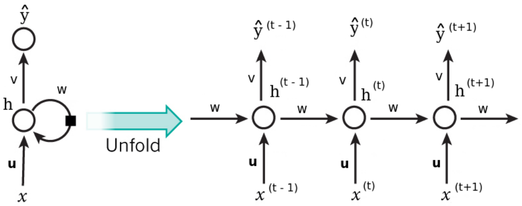
Recurrent Neurons - Weights (3/4)

- $\mathtt{w}$: the weights for the hidden state of the previous time step $\mathtt{h}^{(\mathtt{t}-1)}$.
- $\mathtt{h}^{(\mathtt{t})}$: is the hidden state (memory) at time step $\mathtt{t}$.
  - $\mathtt{h}^{(\mathtt{t})} = \mathtt{tanh}(\mathbf{u}^\intercal \mathbf{x}^{(\mathtt{t})} + \mathtt{w}\mathtt{h}^{(\mathtt{t}-1)})$
  - $\mathtt{h}^{(0)}$ is the initial hidden state.



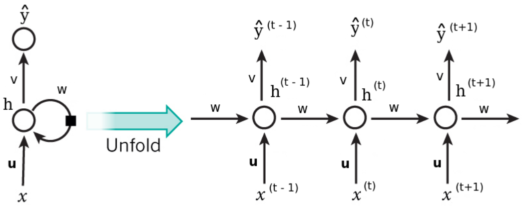
Recurrent Neurons - Weights (4/4)

- $\mathtt{v}$: the weights for the hidden state of the current time step $\mathtt{h}^{(t)}$.



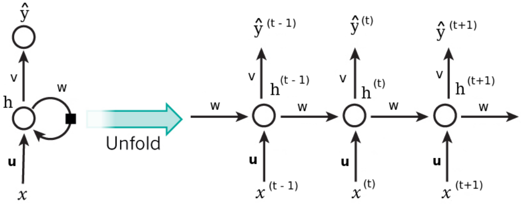
Recurrent Neurons - Weights (4/4)

- $\mathtt{v}$: the weights for the hidden state of the current time step $\mathbf{h}^{(\mathtt{t})}$.
- $\hat{\mathbf{y}}^{(\mathtt{t})}$ is the output at step $\mathtt{t}$.
- $\hat{\mathbf{y}}^{(\mathtt{t})} = \mathtt{softmax}(\mathtt{v}\mathbf{h}^{(\mathtt{t})})$



Recurrent Neurons - Weights (4/4)

- $\mathtt{v}$: the weights for the hidden state of the current time step $\mathtt{h}^{(\mathtt{t})}$.
- $\hat{\mathbf{y}}^{(\mathtt{t})}$ is the output at step $\mathtt{t}$.
- $\hat{\mathbf{y}}^{(\mathtt{t})} = \mathtt{softmax}(\mathtt{vh}^{(\mathtt{t})})$
- For example, if we wanted to predict the next word in a sentence, it would be a vector of probabilities across our vocabulary.

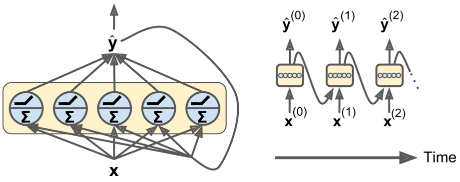


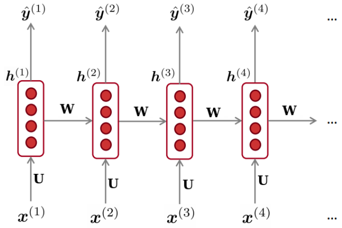
Layers of Recurrent Neurons

- At each time step $\mathtt{t}$, every neuron of a layer receives both the input vector $\mathbf{x}^{(\mathtt{t})}$ and the output vector from the previous time step $\mathbf{h}^{(\mathtt{t}-1)}$.

$$\mathbf{h}^{(\mathtt{t})} = \mathtt{tanh}(\mathbf{u}^\intercal \mathbf{x}^{(\mathtt{t})} + \mathbf{w}^\intercal \mathbf{h}^{(\mathtt{t}-1)})$$

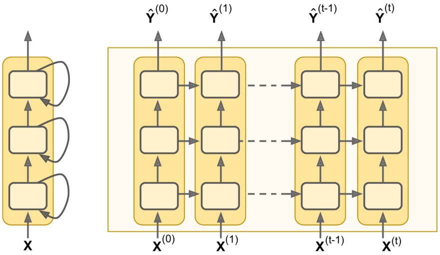
$$\mathbf{y}^{(\mathtt{t})} = \mathtt{sigmoid}(\mathbf{v}^\intercal \mathbf{h}^{(\mathtt{t})})$$





Deep RNN

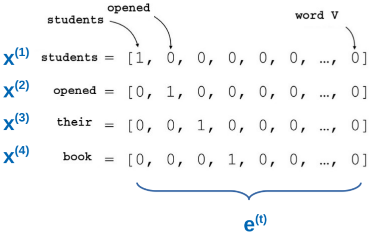
- Stacking multiple layers of cells gives you a deep RNN.

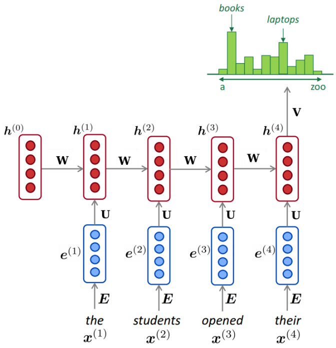


Let's Back to Language Model Example

A RNN Neural Language Model (1/2)

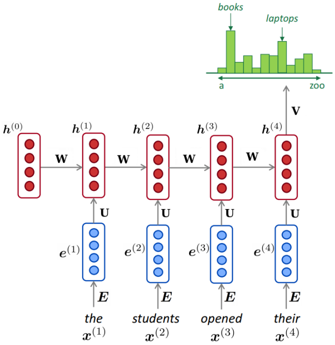
- The input $\mathbf{x}$ will be a sequence of words (each $\mathtt{x}^{(t)}$ is a single word).
- Each embedded word $\mathbf{e}^{(t)} = \mathbf{E}^{\top}\mathtt{x}^{(t)}$ is a one-hot vector of size vocabulary size.





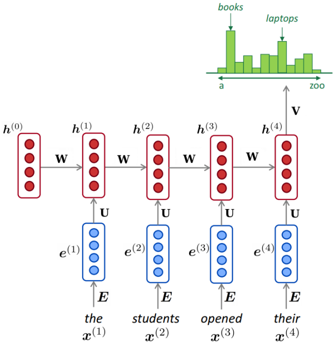
A RNN Neural Language Model (2/2)

- Let's recap the equations for the RNN:
  - $\mathtt{h}^{(\mathtt{t})} = \mathtt{tanh}(\mathbf{u}^{\intercal}\mathbf{e}^{(\mathtt{t})} + \mathtt{wh}^{(\mathtt{t}-1)})$
  - $\hat{\mathbf{y}}^{(\mathtt{t})} = \mathtt{softmax}(\mathtt{vh}^{(\mathtt{t})})$



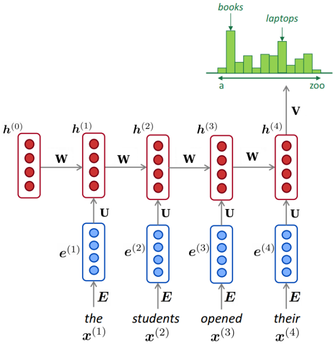
A RNN Neural Language Model (2/2)

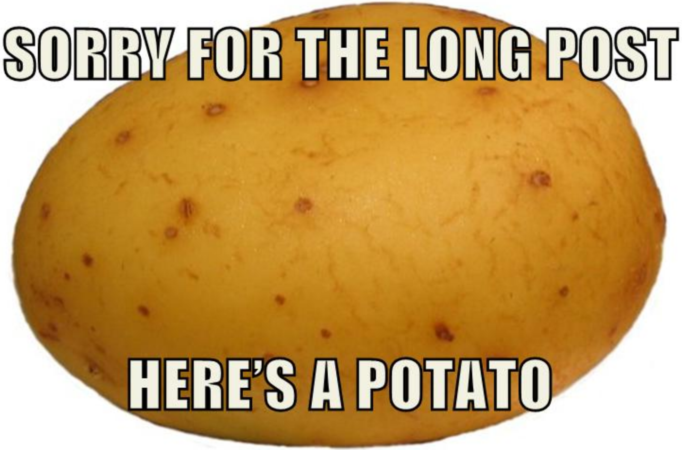
- Let's recap the equations for the RNN:
  - $\mathtt{h}^{(t)} = \mathtt{tanh}(\mathbf{u}^\intercal \mathbf{e}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)})$
  - $\hat{\mathbf{y}}^{(t)} = \mathtt{softmax}(\mathtt{v}\mathtt{h}^{(t)})$
- The output $\hat{\mathbf{y}}^{(t)}$ is a vector of vocabulary size elements.



A RNN Neural Language Model (2/2)

- Let's recap the equations for the RNN:
  - $\mathtt{h}^{(\mathtt{t})} = \mathtt{tanh}(\mathbf{u}^\intercal \mathbf{e}^{(\mathtt{t})} + \mathtt{w}\mathtt{h}^{(\mathtt{t}-1)})$
  - $\hat{\mathbf{y}}^{(\mathtt{t})} = \mathtt{softmax}(\mathtt{v}\mathtt{h}^{(\mathtt{t})})$
- The output $\hat{\mathbf{y}}^{(\mathtt{t})}$ is a vector of vocabulary size elements.
- Each element of $\hat{\mathbf{y}}^{(\mathtt{t})}$ represents the probability of that word being the next word in the sentence.




SORRY FOR THE LONG POST
HERE'S A POTATO

RNN in TensorFlow

RNN in TensorFlow (1/3)

- Manul implementation of an RNN

```
# make the dataset
n_inputs = 3
n_neurons = 5

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
```

RNN in TensorFlow (1/3)

- Manul implementation of an RNN

```
# make the dataset
n_inputs = 3
n_neurons = 5

X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])
```

```
# build the network
Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons], dtype=tf.float32))
Wh = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons], dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

h0 = tf.tanh(tf.matmul(X0, Wx) + b)
h1 = tf.tanh(tf.matmul(h0, Wh) + tf.matmul(X1, Wx) + b)
```

RNN in TensorFlow (2/3)

- Use dynamic_rnn

```
n_inputs = 3
n_neurons = 5
n_steps = 2

X_batch = np.array([
        # t = 0       t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
```

RNN in TensorFlow (2/3)

- Use dynamic_rnn

```
n_inputs = 3
n_neurons = 5
n_steps = 2

X_batch = np.array([
        # t = 0      t = 1
        [[0, 1, 2], [9, 8, 7]], # instance 1
        [[3, 4, 5], [0, 0, 0]], # instance 2
        [[6, 7, 8], [6, 5, 4]], # instance 3
        [[9, 0, 1], [3, 2, 1]], # instance 4
    ])

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
```

```
# build the network
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```

RNN in TensorFlow (3/3)

- Multi-layer RNN

```
layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
  for layer in range(n_layers)]

multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)

outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

states_concat = tf.concat(axis=1, values=states)

logits = tf.layers.dense(states_concat, n_outputs)
```

Training RNNs

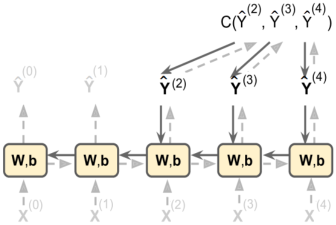
Training RNNs

- To train an RNN, we should unroll it through time and then simply use regular backpropagation.

- This strategy is called backpropagation through time (BPTT).

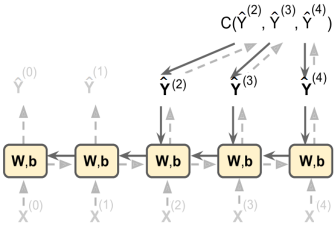
Backpropagation Through Time (1/3)

- To train the model using BPTT, we go through the following steps:
- 1. Forward pass through the unrolled network (represented by the dashed arrows).
- 2. The cost function is $\mathtt{C}(\hat{\mathbf{y}}^{\mathtt{tmin}}, \hat{\mathbf{y}}^{\mathtt{tmin}+1}, \cdots, \hat{\mathbf{y}}^{\mathtt{tmax}})$, where `tmin` and `tmax` are the first and last output time steps, not counting the ignored outputs.



Backpropagation Through Time (2/3)

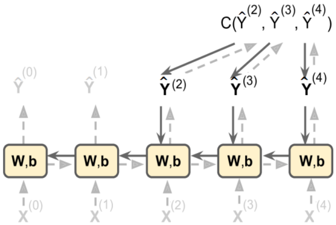
- 3. Propagate backward the gradients of that cost function through the unrolled network (represented by the solid arrows).
- 4. The model parameters are updated using the gradients computed during BPTT.

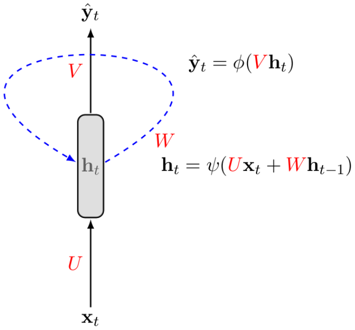


Backpropagation Through Time (3/3)

- The gradients flow backward through all the outputs used by the cost function, not just through the final output.

- For example, in the following figure:
  - The cost function is computed using the last three outputs, $\hat{\mathbf{y}}^{(2)}$, $\hat{\mathbf{y}}^{(3)}$, and $\hat{\mathbf{y}}^{(4)}$.
  - Gradients flow through these three outputs, but not through $\hat{\mathbf{y}}^{(0)}$ and $\hat{\mathbf{y}}^{(1)}$.

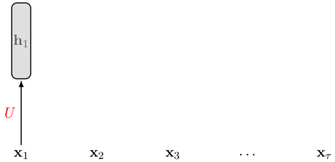



$\hat{\mathbf{y}}_t$
V
$\hat{\mathbf{y}}_t = \phi(V\mathbf{h}_t)$
W
$\mathbf{h}_t$
$\mathbf{h}_t = \psi(U\mathbf{x}_t + W\mathbf{h}_{t-1})$
U
$\mathbf{x}_t$

BPTT Step by Step (2/20)

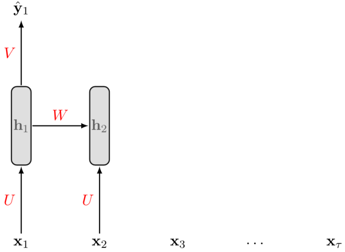
$$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \cdots \quad \mathbf{x}_\tau$$

BPTT Step by Step (3/20)

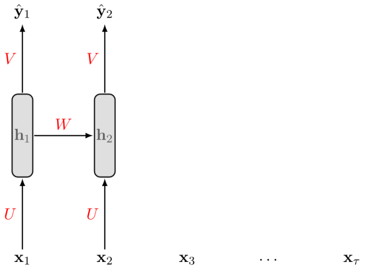


BPTT Step by Step (4/20)

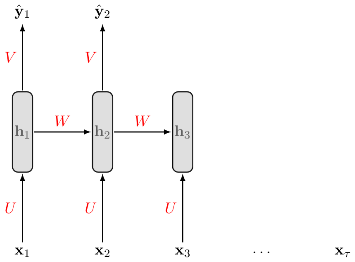
BPTT Step by Step (5/20)



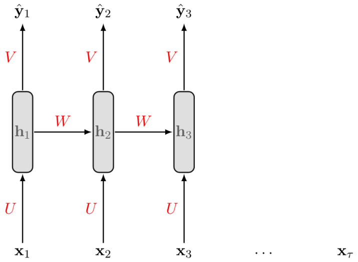
BPTT Step by Step (6/20)



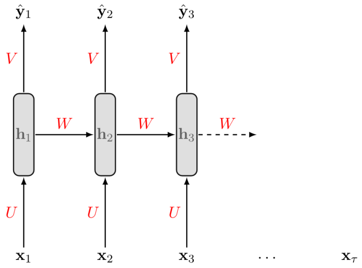
BPTT Step by Step (7/20)



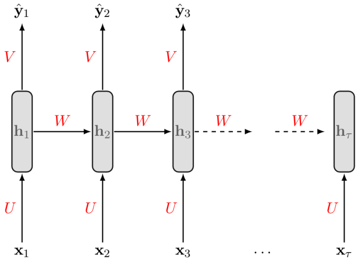
BPTT Step by Step (8/20)



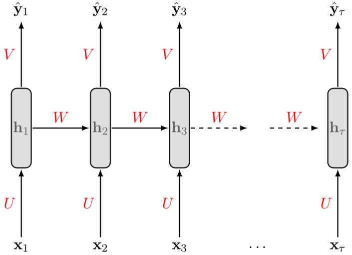
BPTT Step by Step (9/20)



BPTT Step by Step (10/20)



BPTT Step by Step (11/20)



BPTT Step by Step (12/20)

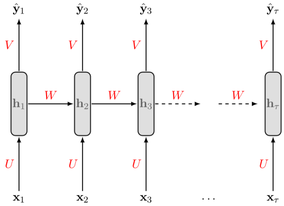
$$\mathbf{s}^{(t)} = \mathbf{u}^T\mathbf{x}^{(t)} + \mathtt{w}\mathtt{h}^{(t-1)}$$

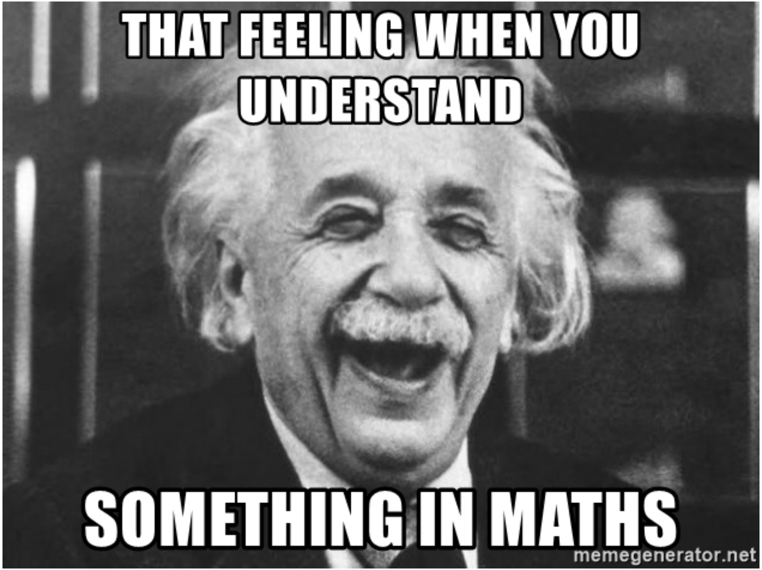
$$\mathtt{h}^{(t)} = \mathtt{tanh}(\mathbf{s}^{(t)})$$

$$\mathtt{z}^{(t)} = \mathtt{v}\mathtt{h}^{(t)}$$

$$\hat{\mathtt{y}}^{(t)} = \mathtt{softmax}(\mathtt{z}^{(t)})$$

$$\mathtt{J}^{(t)} = \mathtt{cross\_entropy}(\mathtt{y}^{(t)}, \hat{\mathtt{y}}^{(t)}) = -\sum \mathtt{y}^{(t)} \mathtt{log} \hat{\mathtt{y}}^{(t)}$$

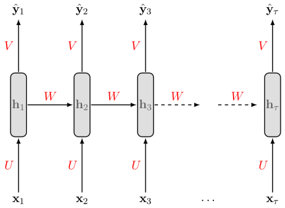



THAT FEELING WHEN YOU UNDERSTAND
SOMETHING IN MATHS
memegenerator.net

BPTT Step by Step (13/20)

$$\mathtt{J}^{(\mathtt{t})} = \mathtt{cross\_entropy}(\mathtt{y}^{(\mathtt{t})}, \hat{\mathtt{y}}^{(\mathtt{t})}) = -\sum \mathtt{y}^{(\mathtt{t})}\mathtt{log}\hat{\mathtt{y}}^{(\mathtt{t})}$$

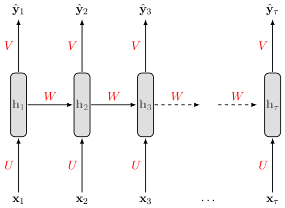
- We treat the full sequence as one training example.



BPTT Step by Step (13/20)

$$\mathtt{J}^{(t)} = \mathtt{cross\_entropy}(\mathtt{y}^{(t)}, \hat{\mathtt{y}}^{(t)}) = -\sum \mathtt{y}^{(t)} \mathtt{log} \hat{\mathtt{y}}^{(t)}$$

- We treat the full sequence as one training example.
- The total error $\mathtt{E}$ is just the sum of the errors at each time step.
- E.g., $\mathtt{E} = \mathtt{J}^{(1)} + \mathtt{J}^{(2)} + \cdots + \mathtt{J}^{(t)}$



BPTT Step by Step (14/20)

- $\mathtt{J}^{(\mathtt{t})}$ is the total cost, so we can say that a 1-unit increase in $\mathtt{v}$, $\mathtt{w}$ or $\mathtt{u}$ will impact each of $\mathtt{J}^{(1)}$, $\mathtt{J}^{(2)}$, until $\mathtt{J}^{(\mathtt{t})}$ individually.

BPTT Step by Step (14/20)

- $\mathtt{J}^{(\mathtt{t})}$ is the total cost, so we can say that a 1-unit increase in $\mathtt{v}$, $\mathtt{w}$ or $\mathtt{u}$ will impact each of $\mathtt{J}^{(1)}$, $\mathtt{J}^{(2)}$, until $\mathtt{J}^{(\mathtt{t})}$ individually.
- The gradient is equal to the sum of the respective gradients at each time step $\mathtt{t}$.

BPTT Step by Step (14/20)

- $\mathtt{J}^{(\mathtt{t})}$ is the total cost, so we can say that a 1-unit increase in $\mathtt{v}$, $\mathtt{w}$ or $\mathtt{u}$ will impact each of $\mathtt{J}^{(1)}$, $\mathtt{J}^{(2)}$, until $\mathtt{J}^{(\mathtt{t})}$ individually.
- The gradient is equal to the sum of the respective gradients at each time step $\mathtt{t}$.
- For example if $\mathtt{t} = 3$ we have: $\mathtt{E} = \mathtt{J}^{(1)} + \mathtt{J}^{(2)} + \mathtt{J}^{(3)}$

BPTT Step by Step (14/20)

- $\mathtt{J}^{(\mathtt{t})}$ is the total cost, so we can say that a 1-unit increase in $\mathtt{v}$, $\mathtt{w}$ or $\mathtt{u}$ will impact each of $\mathtt{J}^{(1)}$, $\mathtt{J}^{(2)}$, until $\mathtt{J}^{(\mathtt{t})}$ individually.
- The gradient is equal to the sum of the respective gradients at each time step $\mathtt{t}$.
- For example if $\mathtt{t} = 3$ we have: $\mathtt{E} = \mathtt{J}^{(1)} + \mathtt{J}^{(2)} + \mathtt{J}^{(3)}$

$$\frac{\partial \mathtt{E}}{\partial \mathtt{v}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{v}}$$

BPTT Step by Step (14/20)

- $J^{(t)}$ is the total cost, so we can say that a 1-unit increase in v, w or u will impact each of $J^{(1)}$, $J^{(2)}$, until $J^{(t)}$ individually.
- The gradient is equal to the sum of the respective gradients at each time step t.
- For example if $t = 3$ we have: $E = J^{(1)} + J^{(2)} + J^{(3)}$

$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

$$\frac{\partial E}{\partial w} = \sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

BPTT Step by Step (14/20)

- $\mathtt{J}^{(\mathtt{t})}$ is the total cost, so we can say that a 1-unit increase in $\mathtt{v}$, $\mathtt{w}$ or $\mathtt{u}$ will impact each of $\mathtt{J}^{(1)}$, $\mathtt{J}^{(2)}$, until $\mathtt{J}^{(\mathtt{t})}$ individually.
- The gradient is equal to the sum of the respective gradients at each time step $\mathtt{t}$.
- For example if $\mathtt{t} = 3$ we have: $\mathtt{E} = \mathtt{J}^{(1)} + \mathtt{J}^{(2)} + \mathtt{J}^{(3)}$

$$\frac{\partial \mathtt{E}}{\partial \mathtt{v}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{v}}$$

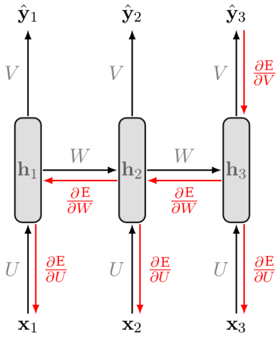
$$\frac{\partial \mathtt{E}}{\partial \mathtt{w}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{w}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{w}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{w}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{w}}$$

$$\frac{\partial \mathtt{E}}{\partial \mathtt{u}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{u}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{u}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{u}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{u}}$$

BPTT Step by Step (15/20)

- Let's start with $\frac{\partial \mathrm{E}}{\partial \mathtt{v}}$.
- A change in $\mathtt{v}$ will only impact $\mathtt{J}^{(3)}$ at time $\mathtt{t} = 3$, because it plays no role in computing the value of anything other than $\mathtt{z}^{(3)}$.

$$\frac{\partial \mathrm{E}}{\partial \mathtt{v}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{v}}$$

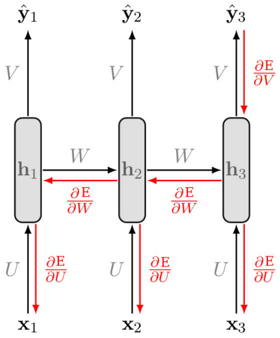


BPTT Step by Step (15/20)

- Let's start with $\frac{\partial \mathtt{E}}{\partial \mathtt{v}}$.
- A change in $\mathtt{v}$ will only impact $\mathtt{J}^{(3)}$ at time $\mathtt{t} = 3$, because it plays no role in computing the value of anything other than $\mathtt{z}^{(3)}$.

$$\frac{\partial \mathtt{E}}{\partial \mathtt{v}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{v}}$$

$$\frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \hat{\mathtt{y}}^{(3)}} \frac{\partial \hat{\mathtt{y}}^{(3)}}{\partial \mathtt{z}^{(3)}} \frac{\partial \mathtt{z}^{(3)}}{\partial \mathtt{v}}$$



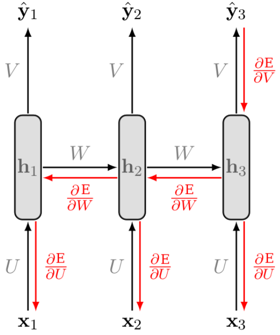
BPTT Step by Step (15/20)

- Let's start with $\frac{\partial \mathrm{E}}{\partial \mathtt{v}}$.
- A change in $\mathtt{v}$ will only impact $\mathtt{J}^{(3)}$ at time $\mathtt{t} = 3$, because it plays no role in computing the value of anything other than $\mathtt{z}^{(3)}$.

$$\frac{\partial \mathrm{E}}{\partial \mathtt{v}} = \sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{v}}$$

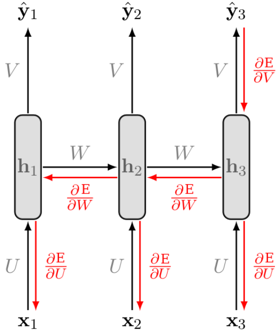
$$\frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \hat{\mathtt{y}}^{(3)}} \frac{\partial \hat{\mathtt{y}}^{(3)}}{\partial \mathtt{z}^{(3)}} \frac{\partial \mathtt{z}^{(3)}}{\partial \mathtt{v}}$$

$$\frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{v}} = \frac{\partial \mathtt{J}^{(2)}}{\partial \hat{\mathtt{y}}^{(2)}} \frac{\partial \hat{\mathtt{y}}^{(2)}}{\partial \mathtt{z}^{(2)}} \frac{\partial \mathtt{z}^{(2)}}{\partial \mathtt{v}}$$



BPTT Step by Step (15/20)

- Let's start with $\frac{\partial E}{\partial v}$.
- A change in v will only impact $J^{(3)}$ at time $t = 3$, because it plays no role in computing the value of anything other than $z^{(3)}$.



$$\frac{\partial E}{\partial v} = \sum_t \frac{\partial J^{(t)}}{\partial v} = \frac{\partial J^{(3)}}{\partial v} + \frac{\partial J^{(2)}}{\partial v} + \frac{\partial J^{(1)}}{\partial v}$$

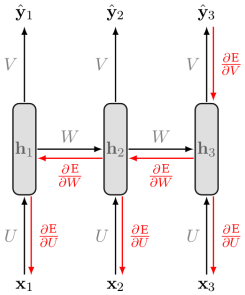
$$\frac{\partial J^{(3)}}{\partial v} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial v}$$

$$\frac{\partial J^{(2)}}{\partial v} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial v}$$

$$\frac{\partial J^{(1)}}{\partial v} = \frac{\partial J^{(1)}}{\partial \hat{y}^{(1)}} \frac{\partial \hat{y}^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial v}$$

BPTT Step by Step (16/20)

- Let's compute the derivatives of $\frac{\partial \mathtt{J}}{\partial \mathtt{w}}$ and $\frac{\partial \mathtt{J}}{\partial \mathtt{u}}$, which are computed the same.
- A change in $\mathtt{w}$ at $\mathtt{t} = 3$ will impact our cost $\mathtt{J}$ in 3 separate ways:
  1. When computing the value of $\mathtt{h}^{(1)}$.
  2. When computing the value of $\mathtt{h}^{(2)}$, which depends on $\mathtt{h}^{(1)}$.
  3. When computing the value of $\mathtt{h}^{(3)}$, which depends on $\mathtt{h}^{(2)}$, which depends on $\mathtt{h}^{(1)}$.

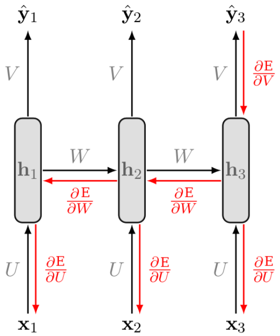


BPTT Step by Step (17/20)

- we compute our individual gradients as:

$$\sum_{\mathtt{t}} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{w}} = \frac{\partial \mathtt{J}^{(3)}}{\partial \mathtt{w}} + \frac{\partial \mathtt{J}^{(2)}}{\partial \mathtt{w}} + \frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{w}}$$

$$\frac{\partial \mathtt{J}^{(1)}}{\partial \mathtt{w}} = \frac{\partial \mathtt{J}^{(1)}}{\partial \hat{\mathtt{y}}^{(1)}} \frac{\partial \hat{\mathtt{y}}^{(1)}}{\partial \mathtt{z}^{(1)}} \frac{\partial \mathtt{z}^{(1)}}{\partial \mathtt{h}^{(1)}} \frac{\partial \mathtt{h}^{(1)}}{\partial \mathtt{s}^{(1)}} \frac{\partial \mathtt{s}^{(1)}}{\partial \mathtt{w}}$$



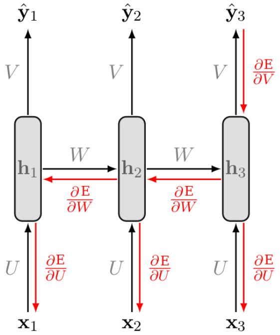
BPTT Step by Step (18/20)

- we compute our individual gradients as:

$$\sum_{t} \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

$$\frac{\partial J^{(2)}}{\partial w} = \frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial w} +$$

$$\frac{\partial J^{(2)}}{\partial \hat{y}^{(2)}} \frac{\partial \hat{y}^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w}$$



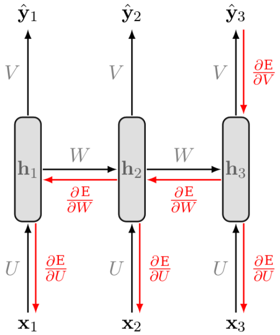
- we compute our individual gradients as:

$$\sum_t \frac{\partial J^{(t)}}{\partial w} = \frac{\partial J^{(3)}}{\partial w} + \frac{\partial J^{(2)}}{\partial w} + \frac{\partial J^{(1)}}{\partial w}$$

$$\frac{\partial J^{(3)}}{\partial w} = \frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial w} +$$

$$\frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial w} +$$

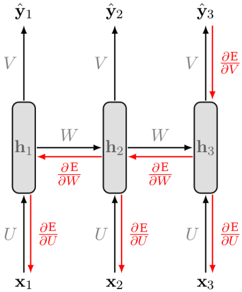
$$\frac{\partial J^{(3)}}{\partial \hat{y}^{(3)}} \frac{\partial \hat{y}^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial s^{(3)}} \frac{\partial s^{(3)}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w}$$



BPTT Step by Step (20/20)

- More generally, a change in $\mathtt{w}$ will impact our cost $\mathtt{J}^{(\mathtt{t})}$ on $\mathtt{t}$ separate occasions.

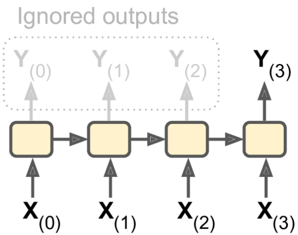
$$\frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \mathtt{w}} = \sum_{k=1}^{t} \frac{\partial \mathtt{J}^{(\mathtt{t})}}{\partial \hat{\mathtt{y}}^{(\mathtt{t})}} \frac{\partial \hat{\mathtt{y}}^{(\mathtt{t})}}{\partial \mathtt{z}^{(\mathtt{t})}} \frac{\partial \hat{\mathtt{z}}^{(\mathtt{t})}}{\partial \mathtt{h}^{(\mathtt{t})}} \left(\prod_{\mathtt{j}=\mathtt{k}+1}^{\mathtt{t}} \frac{\partial \mathtt{h}^{(\mathtt{j})}}{\partial \mathtt{s}^{(\mathtt{j})}} \frac{\partial \mathtt{s}^{(\mathtt{j})}}{\partial \mathtt{h}^{(\mathtt{j}-1)}} \right) \frac{\partial \mathtt{h}^{(\mathtt{k})}}{\partial \mathtt{s}^{(\mathtt{k})}} \frac{\partial \mathtt{s}^{(\mathtt{k})}}{\partial \mathtt{w}}$$



RNN Design Patterns

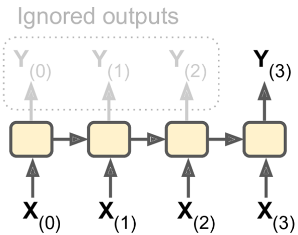
RNN Design Patterns - Sequence-to-Vector

- Sequence-to-vector network: takes a sequence of inputs, and ignore all outputs except for the last one.



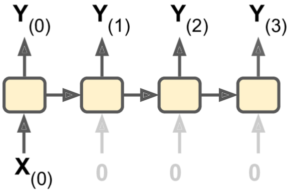
RNN Design Patterns - Sequence-to-Vector

- Sequence-to-vector network: takes a sequence of inputs, and ignore all outputs except for the last one.
- E.g., you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



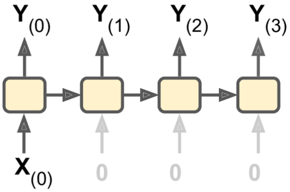
RNN Design Patterns - Vector-to-Sequence

- Vector-to-sequence network: takes a single input at the first time step, and let it output a sequence.



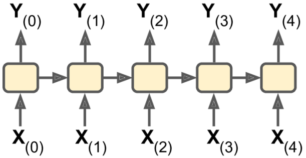
RNN Design Patterns - Vector-to-Sequence

- Vector-to-sequence network: takes a single input at the first time step, and let it output a sequence.
- E.g., the input could be an image, and the output could be a caption for that image.



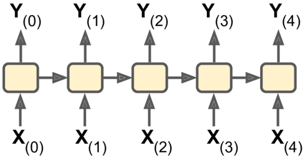
RNN Design Patterns - Sequence-to-Sequence

- Sequence-to-sequence network: takes a sequence of inputs and produce a sequence of outputs.



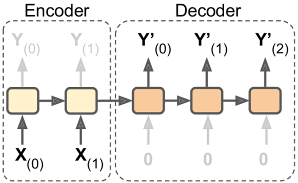
RNN Design Patterns - Sequence-to-Sequence

- Sequence-to-sequence network: takes a sequence of inputs and produce a sequence of outputs.
- Useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future.
- Here, both input sequences and output sequences have the same length.



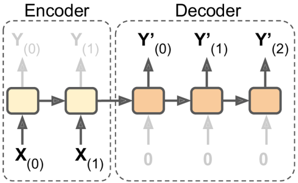
RNN Design Patterns - Encoder-Decoder

- Encoder-decoder network: a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder).



RNN Design Patterns - Encoder-Decoder

- Encoder-decoder network: a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder).
- E.g., translating a sentence from one language to another.
- You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.



LSTM

RNN Problems

- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.

RNN Problems

- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.

RNN Problems

- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
- But, as that gap grows, RNNs become unable to learn to connect the information.
- RNNs may suffer from the vanishing/exploding gradients problem.

RNN Problems

- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
- But, as that gap grows, RNNs become unable to learn to connect the information.
- RNNs may suffer from the vanishing/exploding gradients problem.
- To solve these problem, long short-term memory (LSTM) have been introduced.

RNN Problems

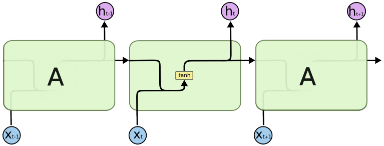
- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.
- But, as that gap grows, RNNs become unable to learn to connect the information.
- RNNs may suffer from the vanishing/exploding gradients problem.
- To solve these problem, long short-term memory (LSTM) have been introduced.
- In LSTM, the network can learn what to store and what to throw away.

RNN Basic Cell vs. LSTM

- Without looking inside the box, the LSTM cell looks exactly like a basic cell.

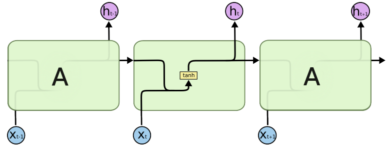
RNN Basic Cell vs. LSTM

- Without looking inside the box, the LSTM cell looks exactly like a basic cell.
- The repeating module in a standard RNN contains a single layer.

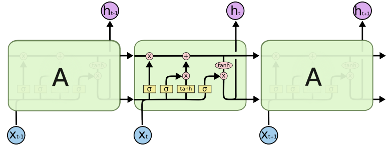


RNN Basic Cell vs. LSTM

- Without looking inside the box, the LSTM cell looks exactly like a basic cell.
- The repeating module in a standard RNN contains a single layer.

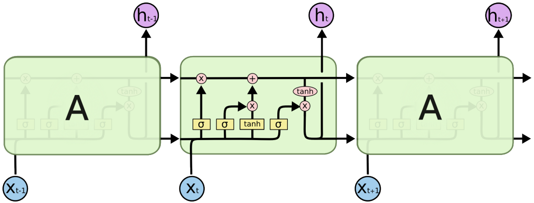


- The repeating module in an LSTM contains four interacting layers.

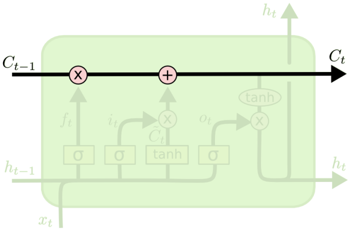


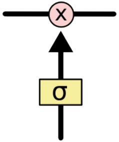
LSTM (1/2)

- In LSTM state is split in two vectors:
  1. $\mathtt{h}^{(t)}$ ($\mathtt{h}$ stands for hidden): the short-term state
  2. $\mathtt{c}^{(t)}$ ($\mathtt{c}$ stands for cell): the long-term state



- The cell state (long-term state), the horizontal line on the top of the diagram.
- The LSTM can remove/add information to the cell state, regulated by three gates.
  - Forget gate, input gate and output gate

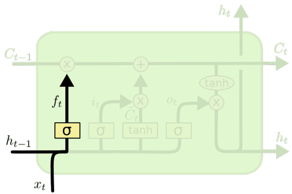




Step-by-Step LSTM Walk Through (1/4)

- Step one: decides what information we are going to throw away from the cell state.

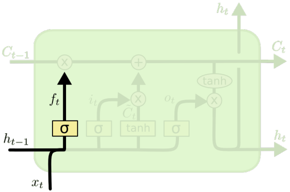
$$\mathtt{f}^{(\mathtt{t})} = \sigma(\mathbf{u}_{\mathtt{f}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\mathtt{f}}\mathtt{h}^{(\mathtt{t}-1)})$$



Step-by-Step LSTM Walk Through (1/4)

- Step one: decides what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer, called the forget gate layer.

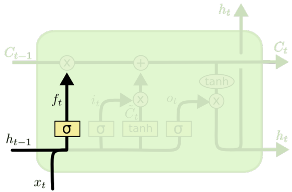
$$\mathtt{f}^{(\mathtt{t})} = \sigma(\mathbf{u}_{\mathtt{f}}^{\mathsf{T}} \mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\mathtt{f}} \mathtt{h}^{(\mathtt{t}-1)})$$



Step-by-Step LSTM Walk Through (1/4)

- Step one: decides what information we are going to throw away from the cell state.
- This decision is made by a sigmoid layer, called the forget gate layer.
- It looks at $\mathtt{h}^{(t-1)}$ and $\mathbf{x}^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $\mathtt{c}^{(t-1)}$.
  - 1 represents completely keep this, and 0 represents completely get rid of this.

$$\mathtt{f}^{(t)} = \sigma(\mathbf{u}_\mathtt{f}^\intercal \mathbf{x}^{(t)} + \mathtt{w}_\mathtt{f} \mathtt{h}^{(t-1)})$$

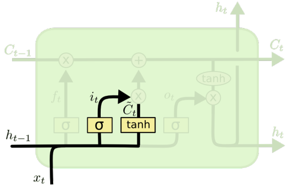


Step-by-Step LSTM Walk Through (2/4)

- Second step: decides what new information we are going to store in the cell state. This has two parts:

$$\mathtt{i}^{(t)} = \sigma(\mathbf{u}_{\mathtt{i}}^{\mathsf{T}}\mathbf{x}^{(t)} + \mathtt{w}_{\mathtt{i}}\mathtt{h}^{(t-1)})$$

$$\tilde{\mathtt{c}}^{(t)} = \tanh(\mathbf{u}_{\tilde{\mathtt{c}}}^{\mathsf{T}}\mathbf{x}^{(t)} + \mathtt{w}_{\tilde{\mathtt{c}}}\mathtt{h}^{(t-1)})$$

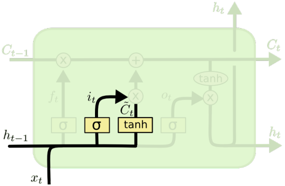


Step-by-Step LSTM Walk Through (2/4)

- Second step: decides what new information we are going to store in the cell state. This has two parts:
- 1. A sigmoid layer, called the input gate layer, decides which values we will update.

$$\mathtt{i}^{(\mathtt{t})} = \sigma(\mathbf{u}_{\mathtt{i}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\mathtt{i}}\mathtt{h}^{(\mathtt{t}-1)})$$

$$\tilde{\mathtt{c}}^{(\mathtt{t})} = \tanh(\mathbf{u}_{\tilde{\mathtt{c}}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\tilde{\mathtt{c}}}\mathtt{h}^{(\mathtt{t}-1)})$$

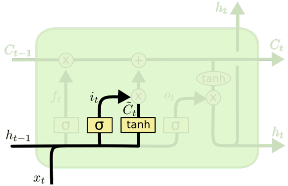


Step-by-Step LSTM Walk Through (2/4)

- Second step: decides what new information we are going to store in the cell state. This has two parts:
- 1. A sigmoid layer, called the input gate layer, decides which values we will update.
- 2. A tanh layer creates a vector of new candidate values that could be added to the state.

$$\mathtt{i}^{(\mathtt{t})} = \sigma(\mathbf{u}_{\mathtt{i}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\mathtt{i}}\mathtt{h}^{(\mathtt{t}-1)})$$

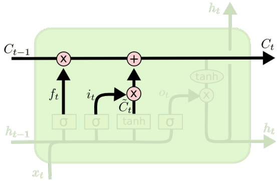
$$\tilde{\mathtt{c}}^{(\mathtt{t})} = \tanh(\mathbf{u}_{\tilde{\mathtt{c}}}^{\mathsf{T}}\mathbf{x}^{(\mathtt{t})} + \mathtt{w}_{\tilde{\mathtt{c}}}\mathtt{h}^{(\mathtt{t}-1)})$$



Step-by-Step LSTM Walk Through (3/4)

- Third step: updates the old cell state $\mathtt{c}^{(t-1)}$, into the new cell state $\mathtt{c}^{(t)}$.

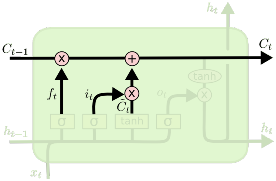
$$\mathtt{c}^{(t)} = \mathtt{f}^{(t)} \otimes \mathtt{c}^{(t-1)} + \mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- Third step: updates the old cell state $\mathtt{c}^{(t-1)}$, into the new cell state $\mathtt{c}^{(t)}$.
- We multiply the old state by $\mathtt{f}^{(t)}$, forgetting the things we decided to forget earlier.

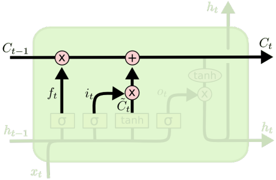
$$\mathtt{c}^{(t)} = \mathtt{f}^{(t)} \otimes \mathtt{c}^{(t-1)} + \mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- Third step: updates the old cell state $\mathtt{c}^{(t-1)}$, into the new cell state $\mathtt{c}^{(t)}$.
- We multiply the old state by $\mathtt{f}^{(t)}$, forgetting the things we decided to forget earlier.
- Then we add it $\mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$.

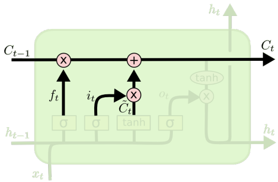
$$\mathtt{c}^{(t)} = \mathtt{f}^{(t)} \otimes \mathtt{c}^{(t-1)} + \mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$$



Step-by-Step LSTM Walk Through (3/4)

- Third step: updates the old cell state $\mathtt{c}^{(t-1)}$, into the new cell state $\mathtt{c}^{(t)}$.
- We multiply the old state by $\mathtt{f}^{(t)}$, forgetting the things we decided to forget earlier.
- Then we add it $\mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$.
- This is the new candidate values, scaled by how much we decided to update each state value.

$$\mathtt{c}^{(t)} = \mathtt{f}^{(t)} \otimes \mathtt{c}^{(t-1)} + \mathtt{i}^{(t)} \otimes \tilde{\mathtt{c}}^{(t)}$$

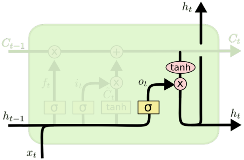


Step-by-Step LSTM Walk Through (4/4)

- Fourth step: decides about the output.

$$\mathtt{o}^{(t)} = \sigma(\mathbf{u}_o^\top \mathbf{x}^{(t)} + \mathtt{w}_o \mathtt{h}^{(t-1)})$$

$$\hat{\mathtt{y}}^{(t)} = \mathtt{h}^{(t)} = \mathtt{o}^{(t)} \otimes \tanh(\mathtt{c}^{(t)})$$

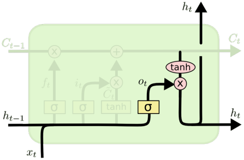


Step-by-Step LSTM Walk Through (4/4)

- Fourth step: decides about the output.
- First, runs a sigmoid layer that decides what parts of the cell state we are going to output.

$$o^{(t)} = \sigma(\mathbf{u}_o^\top \mathbf{x}^{(t)} + \mathtt{w}_o \mathtt{h}^{(t-1)})$$

$$\hat{y}^{(t)} = \mathtt{h}^{(t)} = o^{(t)} \otimes \tanh(c^{(t)})$$

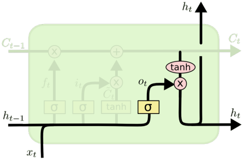


Step-by-Step LSTM Walk Through (4/4)

- Fourth step: decides about the output.
- First, runs a sigmoid layer that decides what parts of the cell state we are going to output.
- Then, puts the cell state through tanh and multiplies it by the output of the sigmoid gate (output gate), so that it only outputs the parts it decided to.

$$\mathtt{o}^{(t)} = \sigma(\mathbf{u}_{\mathtt{o}}^{\mathsf{T}}\mathbf{x}^{(t)} + \mathtt{w}_{\mathtt{o}}\mathtt{h}^{(t-1)})$$

$$\hat{\mathtt{y}}^{(t)} = \mathtt{h}^{(t)} = \mathtt{o}^{(t)} \otimes \tanh(\mathtt{c}^{(t)})$$



LSTM in TensorFlow

- Multi-layer LSTM

```
lstm_cells = [tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons) for layer in range(n_layers)]

multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)

outputs, states = tf.nn.dynamic_rnn(multi_cell, X, dtype=tf.float32)

top_layer_h_state = states[-1][1]

logits = tf.layers.dense(top_layer_h_state, n_outputs)
```

Autoencoders

Let's Start With An Example

- Which of them is easier to memorize?

- Which of them is easier to memorize?
- `Seq1:` $40, 27, 25, 36, 81, 57, 10, 73, 19, 68$

- Which of them is easier to memorize?
- `Seq1:` 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- `Seq2:` 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

$$\mathrm{Seq1}: 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\mathrm{Seq2}: 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

$$\text{Seq1} : 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\text{Seq2} : 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- Seq1 is shorter, so it should be easier.

$$\text{Seq1}: 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\text{Seq2}: 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- Seq1 is shorter, so it should be easier.
- But, Seq2 follows two simple rules:

$$\text{Seq1}: 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\text{Seq2}: 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- Seq1 is shorter, so it should be easier.
- But, Seq2 follows two simple rules:
  - Even numbers are followed by their half.
  - Odd numbers are followed by their triple plus one.

$$\texttt{Seq1} : 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\texttt{Seq2} : 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- `Seq1` is shorter, so it should be easier.
- But, `Seq2` follows two simple rules:
  - Even numbers are followed by their half.
  - Odd numbers are followed by their triple plus one.
- You don't need pattern if you could quickly and easily memorize very long sequences

$$\texttt{Seq1} : 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\texttt{Seq2} : 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- `Seq1` is shorter, so it should be easier.
- But, `Seq2` follows two simple rules:
  - Even numbers are followed by their half.
  - Odd numbers are followed by their triple plus one.
- You don't need pattern if you could quickly and easily memorize very long sequences
- But, it is hard to memorize long sequences that makes it useful to recognize patterns.

- 1970, W. Chase and H. Simon
- They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just 5 seconds.

- This was only the case when the pieces were placed in realistic positions, not when the pieces were placed randomly.

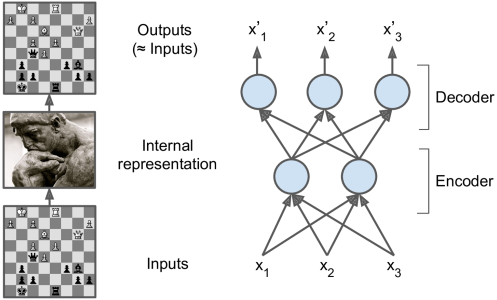
- This was only the case when the pieces were placed in realistic positions, not when the pieces were placed randomly.
- Chess experts don't have a much better memory than you and I.

- This was only the case when the pieces were placed in realistic positions, not when the pieces were placed randomly.
- Chess experts don't have a much better memory than you and I.
- They just see chess patterns more easily due to their experience with the game.

- This was only the case when the pieces were placed in realistic positions, not when the pieces were placed randomly.
- Chess experts don't have a much better memory than you and I.
- They just see chess patterns more easily due to their experience with the game.
- Patterns helps them store information efficiently.

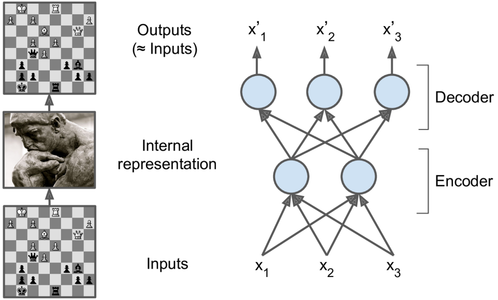
Autoencoders (1/5)

- Just like the chess players in this memory experiment.



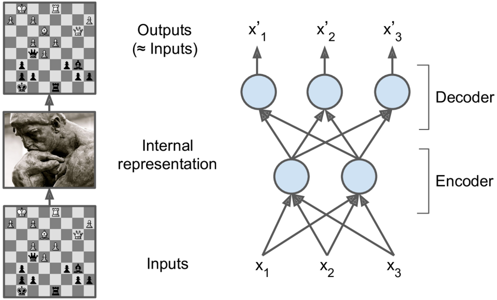
Autoencoders (1/5)

- Just like the chess players in this memory experiment.
- An autoencoder looks at the inputs, converts them to an efficient internal representation, and then spits out something that looks very close to the inputs.



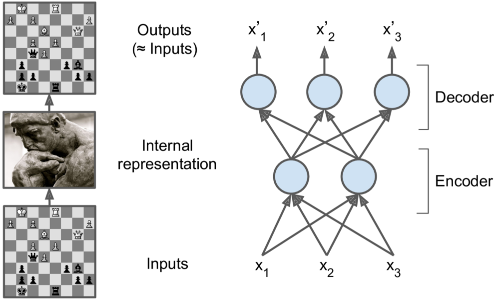
Autoencoders (2/5)

- The same architecture as a Multi-Layer Perceptron (MLP).



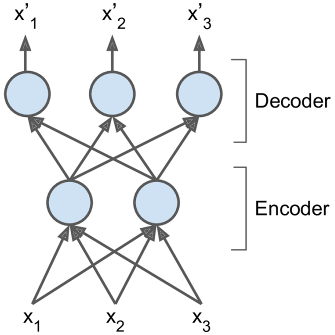
Autoencoders (2/5)

- The same architecture as a Multi-Layer Perceptron (MLP).
- Except that the number of neurons in the output layer must be equal to the number of inputs.



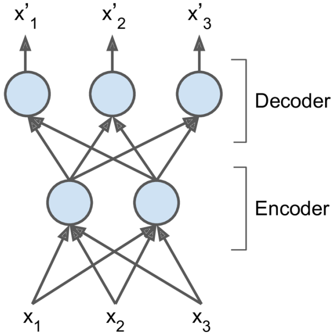
Autoencoders (3/5)

- An autoencoder is always composed of two parts.



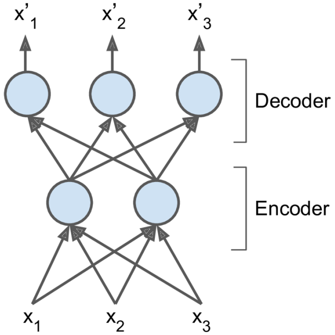
Autoencoders (3/5)

- An autoencoder is always composed of two parts.
- An encoder (recognition network), $\mathbf{h} = \mathtt{f}(\mathbf{x})$
  Converts the inputs to an internal representation.



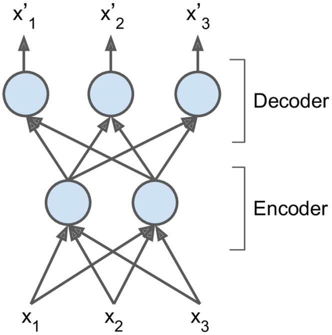
Autoencoders (3/5)

- An autoencoder is always composed of two parts.
- An encoder (recognition network), $\mathbf{h} = \mathtt{f}(\mathbf{x})$
  Converts the inputs to an internal representation.
- A decoder (generative network), $\mathbf{r} = \mathtt{g}(\mathbf{h})$
  Converts the internal representation to the outputs.



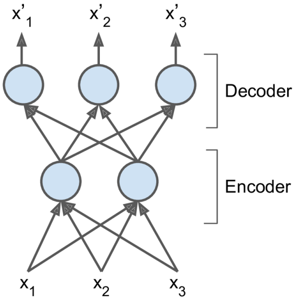
Autoencoders (3/5)

- An autoencoder is always composed of two parts.
- An encoder (recognition network), $\mathbf{h} = \mathtt{f}(\mathbf{x})$
  Converts the inputs to an internal representation.
- A decoder (generative network), $\mathbf{r} = \mathtt{g}(\mathbf{h})$
  Converts the internal representation to the outputs.
- If an autoencoder learns to set $\mathtt{g}(\mathtt{f}(\mathbf{x})) = \mathbf{x}$ everywhere, it is not especially useful, why?



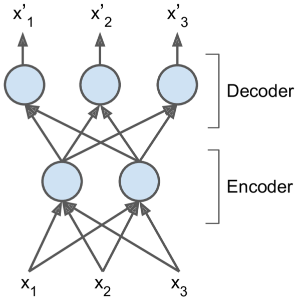
Autoencoders (4/5)

- Autoencoders are designed to be unable to learn to copy perfectly.



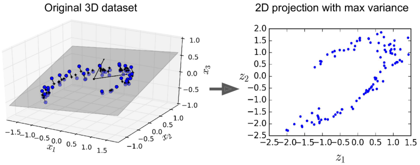
Autoencoders (4/5)

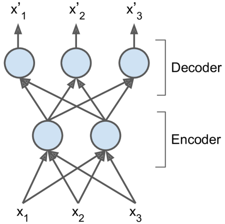
- Autoencoders are designed to be unable to learn to copy perfectly.
- The models are forced to prioritize which aspects of the input should be copied, they often learn useful properties of the data.



Autoencoders (5/5)

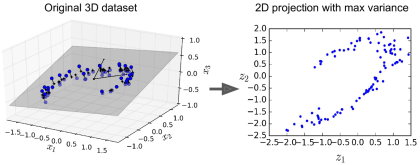
- Autoencoders are neural networks capable of learning efficient representations of the input data (called codings) without any supervision.

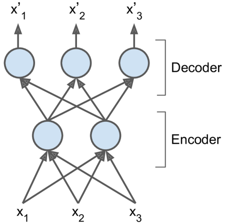




Autoencoders (5/5)

- Autoencoders are neural networks capable of learning efficient representations of the input data (called codings) without any supervision.
- Dimension reduction: these codings typically have a much lower dimensionality than the input data.





Different Types of Autoencoders

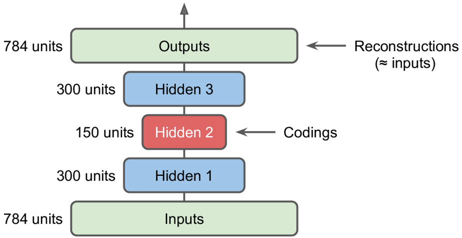
- Stacked autoencoders
- Denoising autoencoders
- Variational autoencoders

Different Types of Autoencoders

- Stacked autoencoders
- Denoising autoencoders
- Variational autoencoders

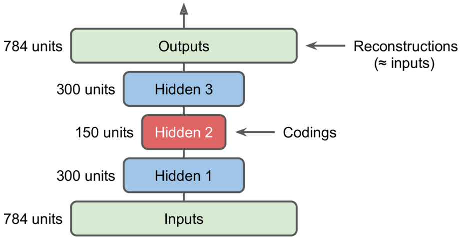
Stacked Autoencoders (1/3)

- Stacked autoencoder: autoencoders with multiple hidden layers.



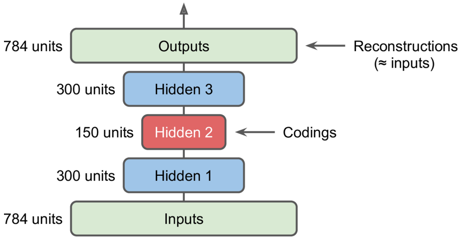
Stacked Autoencoders (1/3)

- Stacked autoencoder: autoencoders with multiple hidden layers.
- Adding more layers helps the autoencoder learn more complex codings.



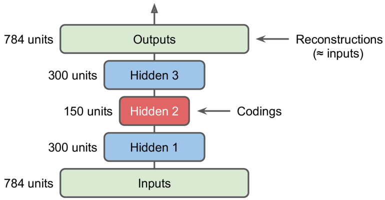
Stacked Autoencoders (1/3)

- Stacked autoencoder: autoencoders with multiple hidden layers.
- Adding more layers helps the autoencoder learn more complex codings.
- The architecture is typically symmetrical with regards to the central hidden layer.



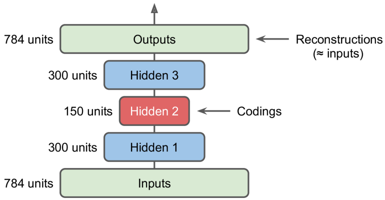
Stacked Autoencoders (2/3)

- In a symmetric architecture, we can tie the weights of the decoder layers to the weights of the encoder layers.



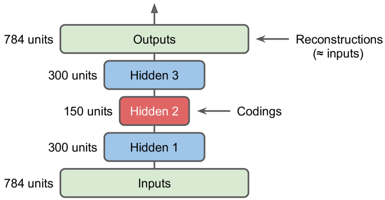
Stacked Autoencoders (2/3)

- In a symmetric architecture, we can tie the weights of the decoder layers to the weights of the encoder layers.
- In a network with N layers, the decoder layer weights can be defined as $w_{N-l+1} = w_l^T$, with $l = 1, 2, \cdots, \frac{N}{2}$.



Stacked Autoencoders (2/3)

- In a symmetric architecture, we can tie the weights of the decoder layers to the weights of the encoder layers.
- In a network with $\mathtt{N}$ layers, the decoder layer weights can be defined as $\mathtt{w}_{\mathtt{N}-\mathtt{l}+1} = \mathtt{w}_{\mathtt{l}}^{\mathtt{T}}$, with $\mathtt{l} = 1, 2, \cdots, \frac{\mathtt{N}}{2}$.
- This halves the number of weights in the model, speeding up training and limiting the risk of overfitting.



Stacked Autoencoders (3/3)

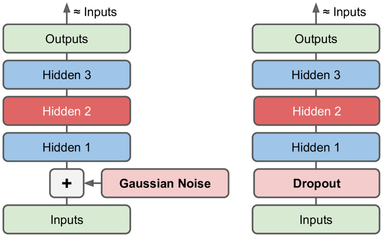
```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 150  # codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

weights1 = tf.Variable(initializer([n_inputs, n_hidden1]), name="weights1")
weights2 = tf.Variable(initializer([n_hidden1, n_hidden2]), name="weights2")
weights3 = tf.transpose(weights2, name="weights3") # tied weights
weights4 = tf.transpose(weights1, name="weights4") # tied weights

hidden1 = tf.nn.elu(tf.matmul(X, weights1) + biases1)
hidden2 = tf.nn.elu(tf.matmul(hidden1, weights2) + biases2)
hidden3 = tf.nn.elu(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4
```

Different Types of Autoencoders

- Stacked autoencoders
- Denoising autoencoders
- Variational autoencoders

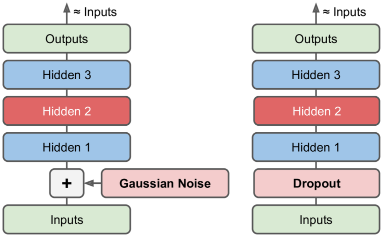
Denoising Autoencoders (1/3)

- One way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original noise-free inputs.



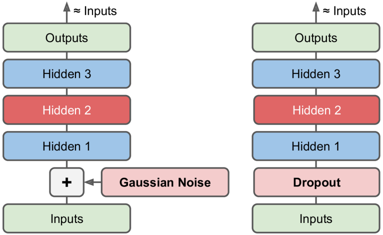
Denoising Autoencoders (1/3)

- One way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original noise-free inputs.
- This prevents the autoencoder from trivially copying its inputs to its outputs, so it ends up having to find patterns in the data.



Denoising Autoencoders (2/3)

- The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched off inputs, just like in dropout.



Denoising Autoencoders (3/3)

```
n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 150  # codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + noise_level * tf.random_normal(tf.shape(X))

hidden1 = tf.layers.dense(X_noisy, n_hidden1, activation=tf.nn.relu, name="hidden1")
hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu, name="hidden2")
hidden3 = tf.layers.dense(hidden2, n_hidden3, activation=tf.nn.relu, name="hidden3")
outputs = tf.layers.dense(hidden3, n_outputs, name="outputs")
```

Different Types of Autoencoders

- Stacked autoencoders
- Denoising autoencoders
- Variational autoencoders

Variational Autoencoders (1/3)

- Variational autoencoders are probabilistic autoencoders.

Variational Autoencoders (1/3)

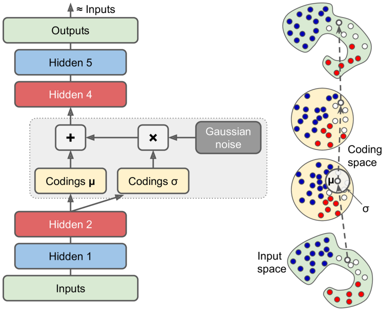
- Variational autoencoders are probabilistic autoencoders.
- Their outputs are partly determined by chance, even after training.
  - As opposed to denoising autoencoders, which use randomness only during training.

Variational Autoencoders (1/3)

- Variational autoencoders are probabilistic autoencoders.
- Their outputs are partly determined by chance, even after training.
  - As opposed to denoising autoencoders, which use randomness only during training.
- They are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.

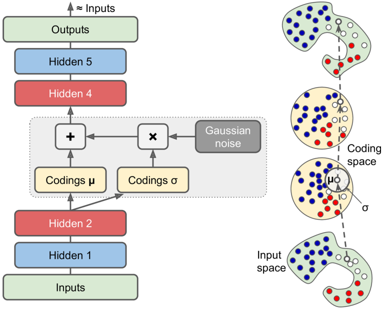
Variational Autoencoders (2/3)

- Instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ.



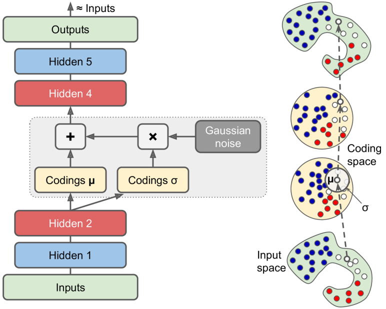
Variational Autoencoders (2/3)

- Instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ.
- The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ.



Variational Autoencoders (2/3)

- Instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ.
- The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ.
- After that the decoder just decodes the sampled coding normally.



Variational Autoencoders (3/3)

- The cost function is composed of two parts.

Variational Autoencoders (3/3)

- The cost function is composed of two parts.
- 1. the usual reconstruction loss.
  - Pushes the autoencoder to reproduce its inputs.
  - Using cross-entropy.

Variational Autoencoders (3/3)

- The cost function is composed of two parts.
- 1. the usual reconstruction loss.
  - Pushes the autoencoder to reproduce its inputs.
  - Using cross-entropy.
- 2. the latent loss
  - Pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution.

Variational Autoencoders (3/3)

- The cost function is composed of two parts.
- 1. the usual reconstruction loss.
  - Pushes the autoencoder to reproduce its inputs.
  - Using cross-entropy.
- 2. the latent loss
  - Pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution.
  - Using the KL divergence between the target distribution (the Gaussian distribution) and the actual distribution of the codings.

Variational Autoencoders (3/3)

- The cost function is composed of two parts.
- 1. the usual reconstruction loss.
  - Pushes the autoencoder to reproduce its inputs.
  - Using cross-entropy.
- 2. the latent loss
  - Pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution.
  - Using the KL divergence between the target distribution (the Gaussian distribution) and the actual distribution of the codings.
  - KL divergence measures the divergence between the two probabilities.

Summary

Summary

- Receptive fields and filters
- Convolution operation
- Padding and strides
- Pooling layer
- Flattening, dropout, dense

Summary

- RNN
- Unfolding the network
- Three weights
- Backpropagation through time
- RNN design patterns
- LSTM

Questions?