



BST 261: Data Science II

Lecture 4

**MLPs IMDb example,
Introduction to Convolutional Neural Networks
(CNNs)**

**Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2020**



Recipe of the day

Brought to you by Erin Lake!

Serious comfort food! - Erin Lake

Peanut Butter Kiss Cookies

Deliciously simple, fast, and easy! Makes 18 cookies (9 servings).

Printed from **Allrecipes**, Submitted by **Sharon Baetcke**

1 cup white sugar	18 milk chocolate candy kisses, unwrapped
1 cup peanut butter	
1 egg	

or 3-ish choc. chips or a few choc. chunks

Directions

- 1 Preheat oven to 350 degrees F.
- 2 Combine sugar, peanut butter, and egg.
- 3 Shape into 1 inch balls and place on ungreased cookie sheet. NOTE: If dough is too sticky, refrigerate 1/2 hour or until easy to handle.
- 4 Bake for 10 minutes. Remove cookies from oven. Press a chocolate kiss into the center of each warm cookie.

only 10 minutes!



IMDb Data Example

The IMDb logo is a yellow rounded rectangle with the letters "IMDb" in bold black font. The background of the slide features a faint, light blue network graph with nodes and connecting lines.

The [IMDb data set](#) is a set of movie reviews that have been labeled as either positive or negative, based on the text content of the reviews

- ⦿ Training set: 25,000 either positive or negative movie reviews that have each been turned into a vector of integers
 - We'll see how to actually do this later in the course
 - Each review can be of any length
 - Only the top 10,000 most frequently occurring words are kept i.e. rare words are discarded
 - Each review includes a label: 0 = negative review and 1 = positive review
- ⦿ Testing set: 25,000 either positive or negative movie reviews, similar to the training set

IMDb Data Example

Data Wrangling

- ◎ Each review is of a varying length and is a list of integers - we need to turn this into a tensor with a common length for each review
- ◎ Create a 2D tensor of shape 25,000 x 10,000
 - 25,000 reviews and 10,000 possible words
- ◎ Use the **vectorize_sequences** function to turn a movie review list of integers into a vector of length 10,000 with 1s for each word that appears in the review and 0s for words that do not
- ◎ The labels are already 0s and 1s, so the only thing we need to do is make them float numbers

Activation and Loss Function Choices

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

IMDb Data Example

Network Architecture

- ◎ 3 layers
 - 2 hidden layers and 1 output layer
 - Hidden layers have 16 hidden units each and a **relu activation function**
 - Output layer has 1 unit (the probability a review is positive)
- ◎ **Sigmoid activation function**
- ◎ **rmsprop optimization algorithm**
- ◎ **binary_crossentropy loss function**
- ◎ **accuracy performance measure** (proportion of times the correct class is chosen)

IMDb Data Example

[Colab link](#)

The background of the slide is a light gray network pattern. It consists of numerous small circles, some of which are solid gray and others are hollow with a gray outline. These circles are interconnected by a web of thin, light gray lines, creating a complex, interconnected mesh that covers the entire slide area.

Regularization

Regularization

- ◎ One of the biggest problems with neural networks is overfitting.
- ◎ Regularization schemes combat overfitting in a variety of different ways

A perceptron represents the following optimization problem:

$$\operatorname{argmin}_W l(y, f(X)) \text{ where } f(X) = \frac{1}{1 + \exp(-\phi(XW))} \text{ (for example)}$$

Regularization

One way to regularize is to introduce penalties and change

$$\operatorname{argmin}_W l(y, f(X))$$

to

$$\operatorname{argmin}_W l(y, f(X)) + \lambda R(W)$$

where $R(W)$ is often the L1 or L2 norm of W . These are the well-known ridge and LASSO penalties, and referred to as **weight decay** by the neural net community.

L2 Regularization

We can limit the size of the L2 norm of the weight vector:

$$\operatorname{argmin}_W l(y, f(X)) + \lambda \|W\|_2$$

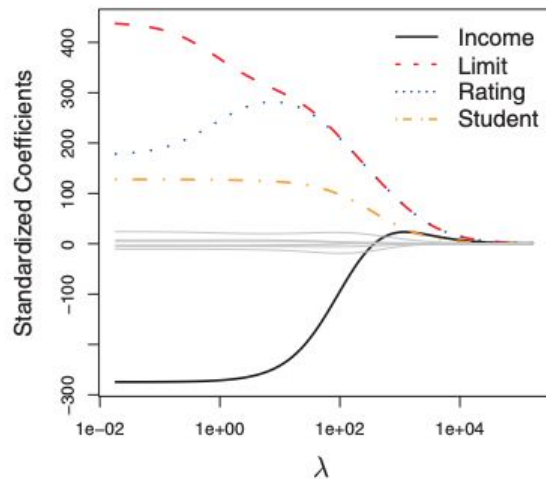
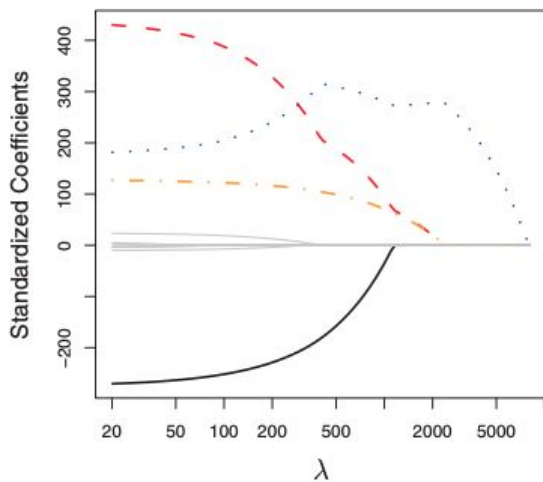
where

$$\|W\|_2 = \sum_{j=1}^p w_j^2$$

We can do the same for the L1 norm. What do the penalties do?

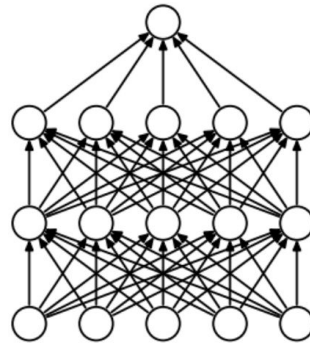
Shrinkage

The L1 and L2 penalties shrink the weights to or towards 0.

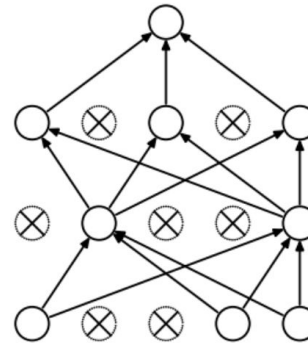


Stochastic Regularization

- Why is this a good idea?
- One of the most popular ways to do this is **dropout**
- Given a hidden layer, we are going to set each element of the hidden layer to 0 with probability p each SGD update.



(a) Standard Neural Net



(b) After applying dropout.

Stochastic Regularization

- One way to think of this is the network is trained by bagged versions of the network.
- Bagging** reduces variance.
- Others have argued this is an approximate Bayesian model

Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning

Yarin Gal
Zoubin Ghahramani
University of Cambridge

YG279@CAM.AC.UK
ZG201@CAM.AC.UK

Abstract

Deep learning tools have gained tremendous attention in applied machine learning. However such tools for regression and classification do not capture model uncertainty. In comparison, Bayesian models offer a mathematically grounded framework to reason about model uncertainty, but usually come with a prohibitive computational cost. In this paper we develop a new theoretical framework casting dropout training in deep neural networks (NNs) as approximate Bayesian inference in deep Gaussian processes. A direct result of this theory gives us tools to model uncertainty with dropout NNs – extracting information from existing models that has been thrown away so far. This mitigates the problem of representing uncertainty in deep

With the recent shift in many of these fields towards the use of Bayesian uncertainty (Herzog & Ostwald, 2013; Trafimow & Marks, 2015; Nuzzo, 2014), new needs arise from deep learning tools.

Standard deep learning tools for regression and classification do not capture model uncertainty. In classification, predictive probabilities obtained at the end of the pipeline (the softmax output) are often erroneously interpreted as model confidence. A model can be uncertain in its predictions even with a high softmax output (fig. 1). Passing a point estimate of a function (solid line 1a) through a softmax (solid line 1b) results in extrapolations with unjustified high confidence for points far from the training data. x^* for example would be classified as class 1 with probability 1. However, passing the distribution (shaded area 1a) through a softmax (shaded area 1b) better reflects classification uncertainty far from the training data.

Stochastic Regularization

- Many have argued that SGD itself provides regularization

Journal of Machine Learning Research 18 (2017) 1-35

Submitted 4/17; Revised 10/17; Published 12/17

Stochastic Gradient Descent as Approximate Bayesian Inference

Stephan Mandt

*Data Science Institute
Department of Computer Science
Columbia University
New York, NY 10025, USA*

SelectorGadget
Has access to this site

STEPHAN.MANDT@GMAIL.COM

Matthew D. Hoffman

*Adobe Research
Adobe Systems Incorporated
601 Townsend Street
San Francisco, CA 94103, USA*

MATHOFFM@ADOBE.COM

David M. Blei

*Department of Statistics
Department of Computer Science
Columbia University
New York, NY 10025, USA*

DAVID.BLEI@COLUMBIA.EDU

Editor: Manfred Opper

Abstract

Stochastic Gradient Descent with a constant learning rate (constant SGD) simulates a Markov chain with a stationary distribution. With this perspective, we derive several new results. (1) We show that constant SGD can be used as an approximate Bayesian posterior inference algorithm. Specifically, we show how to adjust the tuning parameters of constant SGD to best match the stationary distribution to a posterior, minimizing the Kullback-Leibler divergence between these two distributions.

Initialization Regularization

- ◎ The weights in a neural network are given random values initially.
- ◎ There is an entire literature on the best way to do this initialization
 - Normal
 - Truncated Normal
 - Uniform
 - Orthogonal
 - Scaled by number of connections
 - Etc.
- ◎ Try to “bias” the model into initial configurations that are easier to train

Initialization Regularization

- ◎ A popular way is to do **transfer learning**

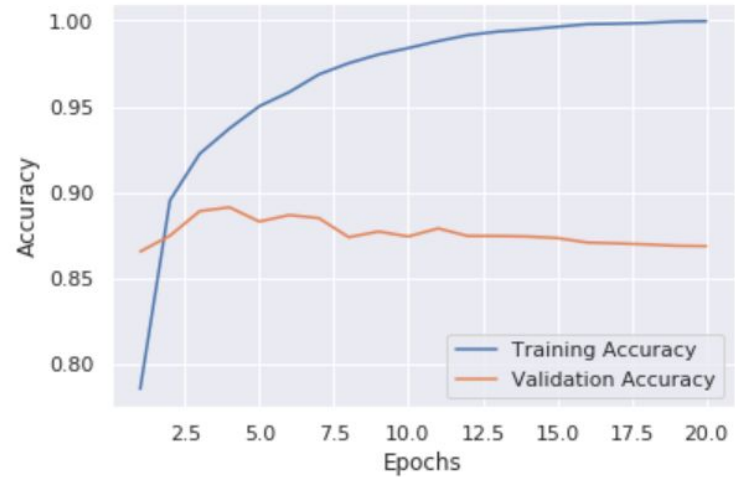
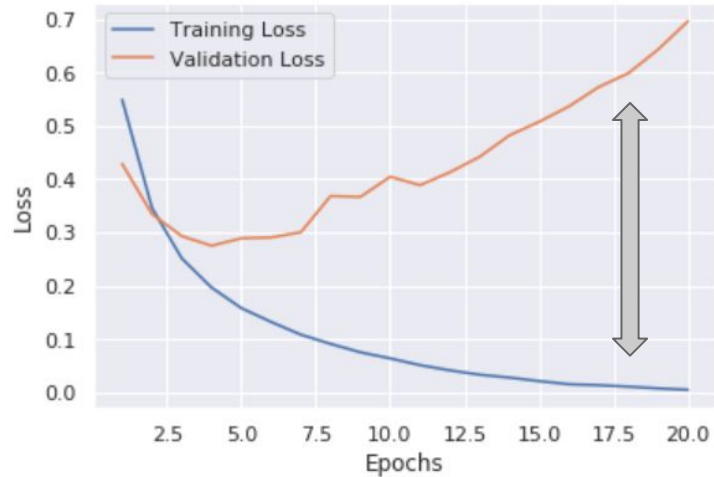
Train the model on auxiliary task
where lots of data is available



Use final weight values from previous
task as initial values and “fine tune”
on primary task

IMDb Example

We saw overfitting in the IMDb example:





How do we make this model better?

Regularization

1. Reduce network size
2. Weight regularization
3. Dropout

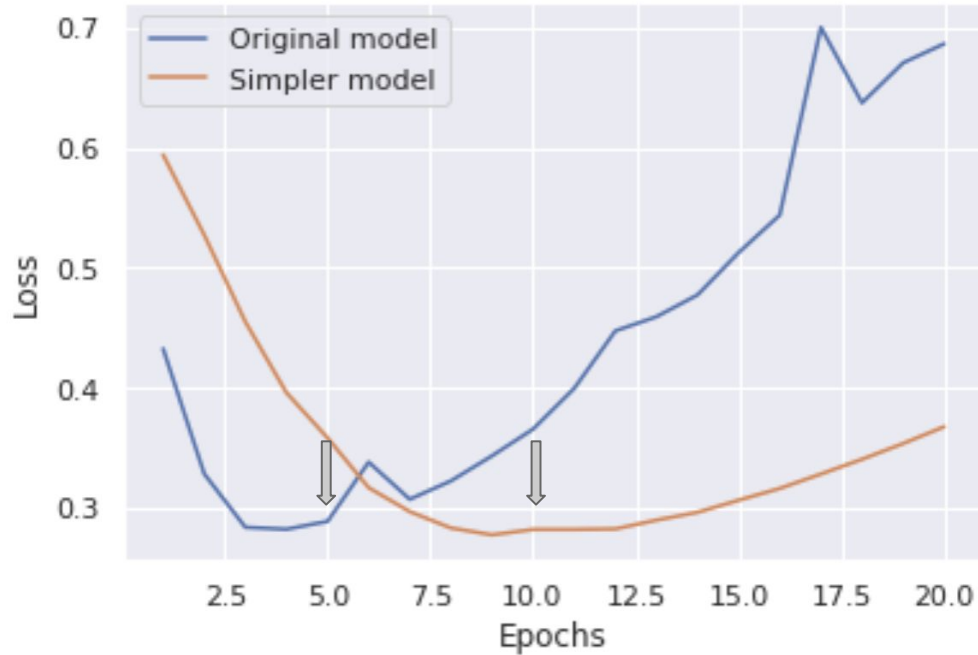
[Back to the IMDb colab](#)

Regularization: reducing network size

When we are battling overfitting, one option is to simplify the model. Let's compare the performance we get from a simpler model. Here we have simplified the model by reducing the number of hidden units in each hidden layer.

```
1 # Original model
2 model = tf.keras.models.Sequential([
3     tf.keras.layers.Dense(16, activation='relu'),
4     tf.keras.layers.Dense(16, activation='relu'),
5     tf.keras.layers.Dense(1, activation='sigmoid')
6 ])
7
8 # Reduced model
9 model2 = tf.keras.models.Sequential([
10     tf.keras.layers.Dense(4, activation='relu'),
11     tf.keras.layers.Dense(4, activation='relu'),
12     tf.keras.layers.Dense(1, activation='sigmoid')
13 ])
```

Regularization: reducing network size

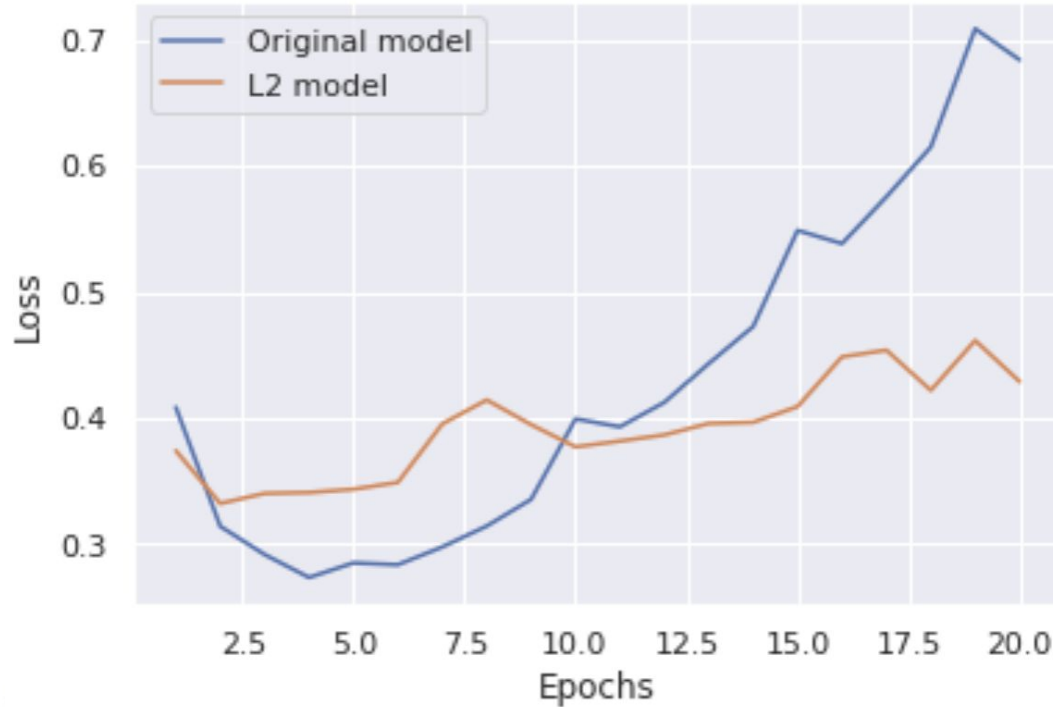


The smaller network performs better than the original model - it starts to overfit at epoch 10 rather than epoch 6. These values are when the validation loss starts to increase.

Regularization: weight regularization

```
1 # L2 model
2 l2_model = tf.keras.models.Sequential([
3     # Layer 1 (Hidden layer)
4     tf.keras.layers.Dense(16, activation='relu',
5                             kernel_regularizer = tf.keras.regularizers.l2(0.001)), ←
6     # Layer 2 (Hidden layer)
7     tf.keras.layers.Dense(16, activation='relu',
8                             kernel_regularizer = tf.keras.regularizers.l2(0.001)), ←
9     # Layer 3 (Output layer)
10    tf.keras.layers.Dense(1, activation='sigmoid')
11 ])
12
13 # Define how to execute training
14 l2_model.compile(optimizer='rmsprop',
15                  loss='binary_crossentropy',
16                  metrics=['accuracy'])
```

Regularization: weight regularization



The L2-regularized model is much more resistant to overfitting - the validation loss starts to increase at a much slower rate

Regularization: adding dropout

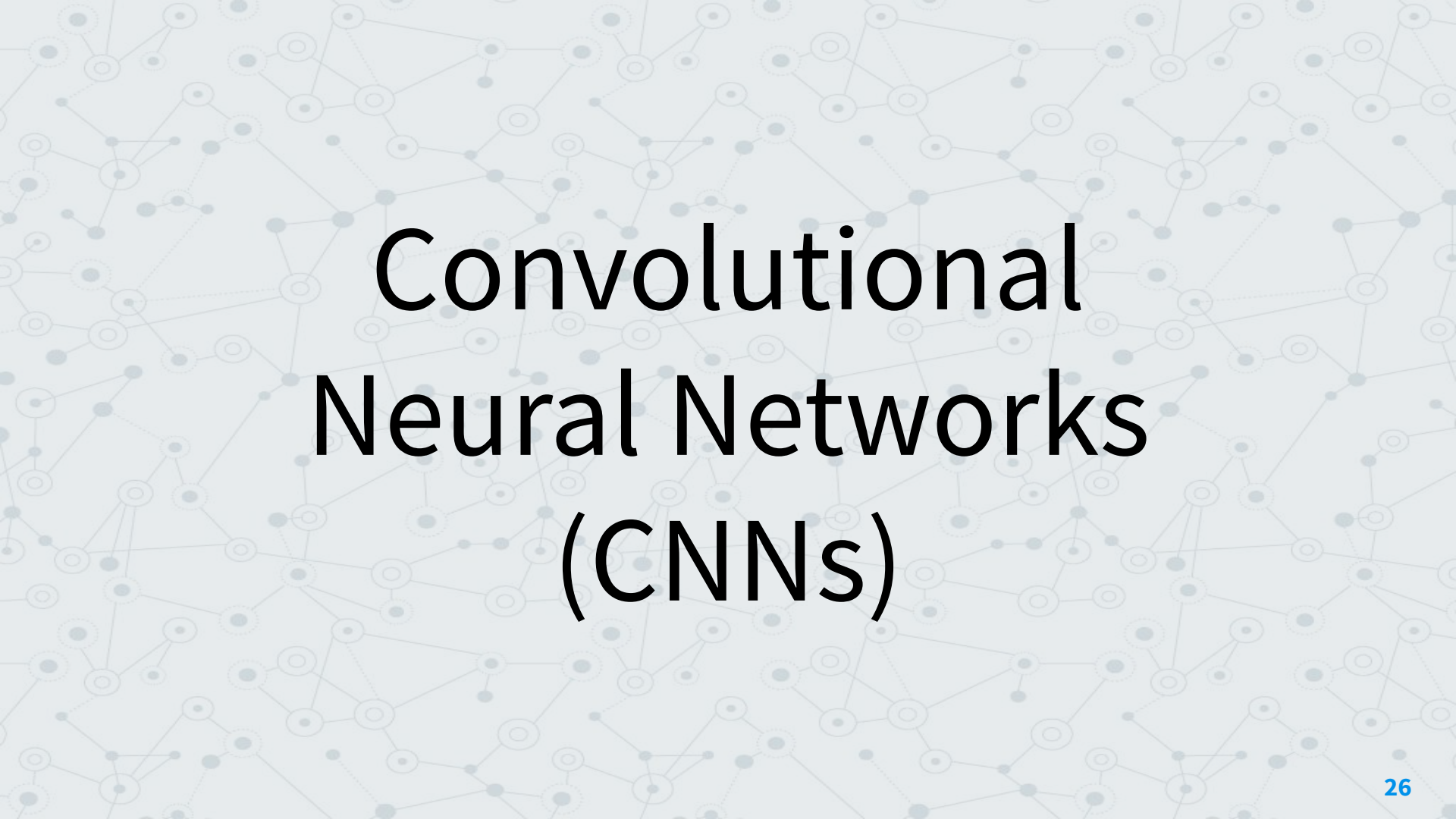
```
1 # Dropout model
2 dmodel = tf.keras.models.Sequential([
3     # Layer 1 (Hidden layer)
4     tf.keras.layers.Dense(16, activation='relu'),
5     # Dropout layer
6     tf.keras.layers.Dropout(0.5), ←
7     # Layer 2 (Hidden layer)
8     tf.keras.layers.Dense(16, activation='relu'),
9     # Dropout layer
10    tf.keras.layers.Dropout(0.5), ←
11    # Layer 3 (Output layer)
12    tf.keras.layers.Dense(1, activation='sigmoid')
13 ])
14
15 # Define how to execute training
16 dmodel.compile(optimizer='rmsprop',
17                loss='binary_crossentropy',
18                metrics=['accuracy'])
```

The 0.5 indicates a 50% probability of dropping out a unit. Typically, 20% is used in practice but you can try different values and see what performs best.

Regularization: adding dropout



The dropout model is slightly better than the original model but does not control for overfitting as well as the L2 network

The background of the slide features a complex, light blue network pattern. It consists of numerous small circles, some solid and some hollow, connected by thin, intersecting lines that form a web-like structure across the entire page.

Convolutional Neural Networks (CNNs)

CNNs

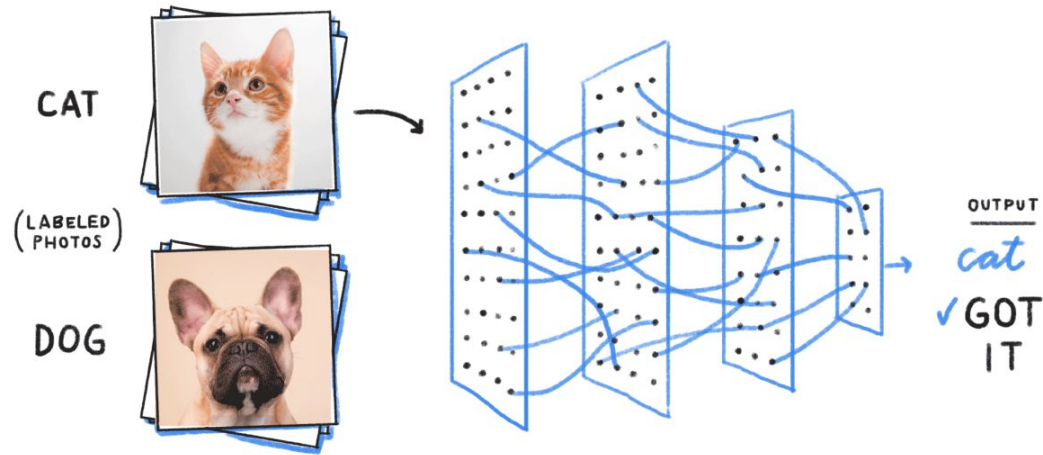
Convolutional neural networks = CNNs = convnets = (many times) computer vision

CNNs are at the heart of deep learning, emerging in recent years as the most prominent strain of neural networks in research. They have revolutionized computer vision, achieving state-of-the-art results in many fundamental tasks, as well as making strong progress in natural language processing, computer vision, reinforcement learning, and many other areas. CNNs have been widely deployed by tech companies for many of the new services and features we see today. They have numerous and diverse applications, including:

- Detecting and labeling objects, locations, and people in images
- Converting speech into text and synthesizing audio of natural sounds
- Describing images and videos with natural language
- Tracking roads and navigating around obstacles in autonomous vehicles
- Analyzing video game screens to guide autonomous agents playing them

CNN Applications

Image classification



CNN Applications

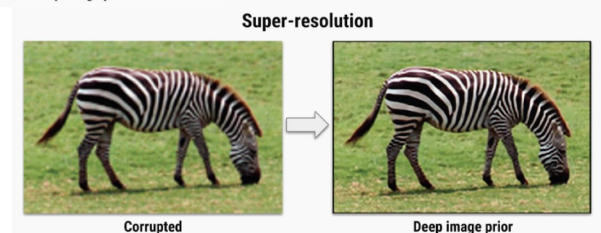
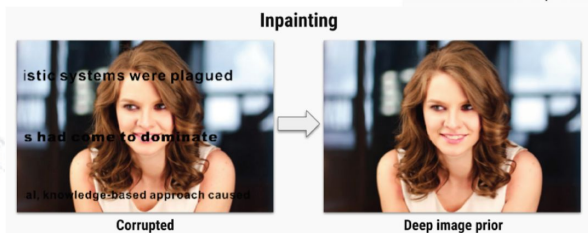
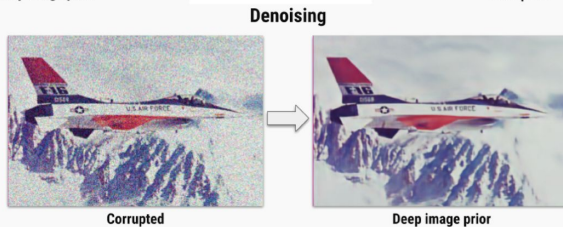
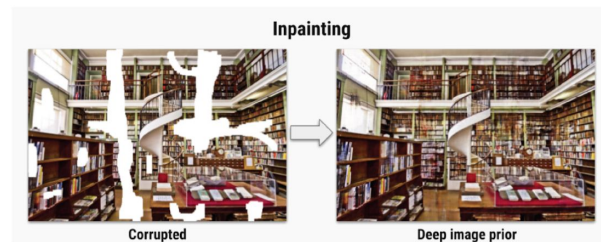
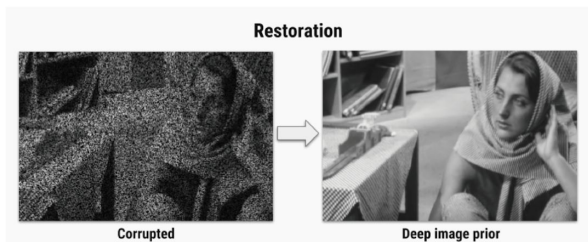
Computer vision



Top: 4 correctly classified examples. Bottom: 4 incorrectly classified examples. Each example has an image, followed by its label, followed by the top 5 guesses with probabilities. From Krizhevsky *et al.* (2012).

CNN Applications

Restoration, inpainting, denoising and super-resolution



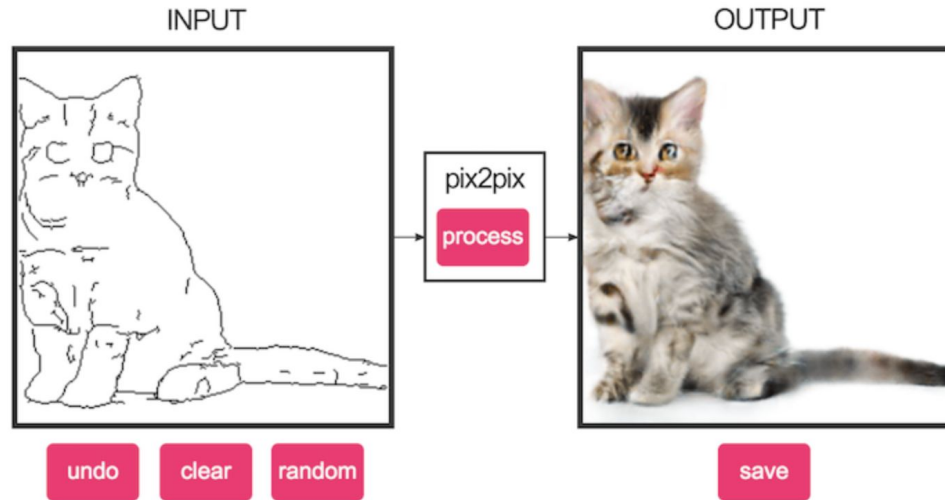
CNN Applications

Self-driving cars



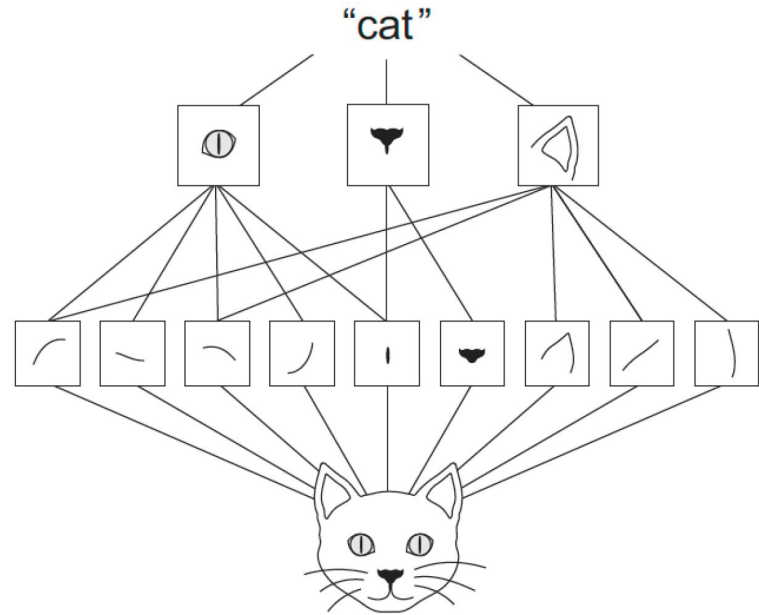
CNN Applications

Generating realistic pictures from drawn edges



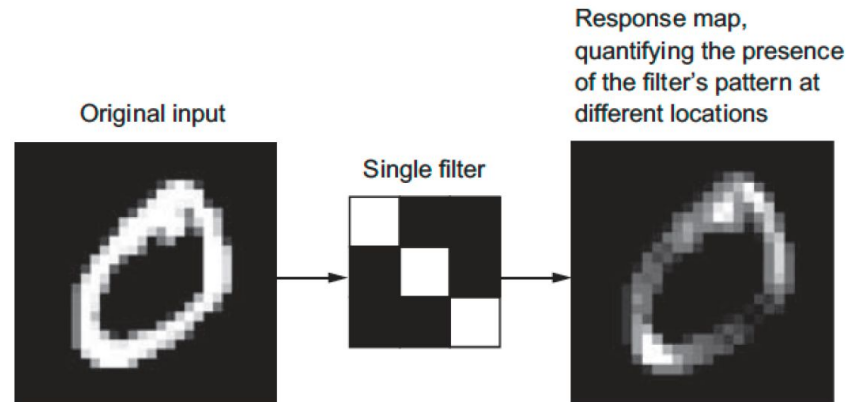
Dense vs Convolutional Layers

- ◎ Dense layers learn **global** patterns
- ◎ Convolutional layers learn **local** patterns
- ◎ CNNs learn **spatial hierarchies of patterns**: one convolutional layer will learn small patterns and the next larger patterns made of the features of the layer before, and so on

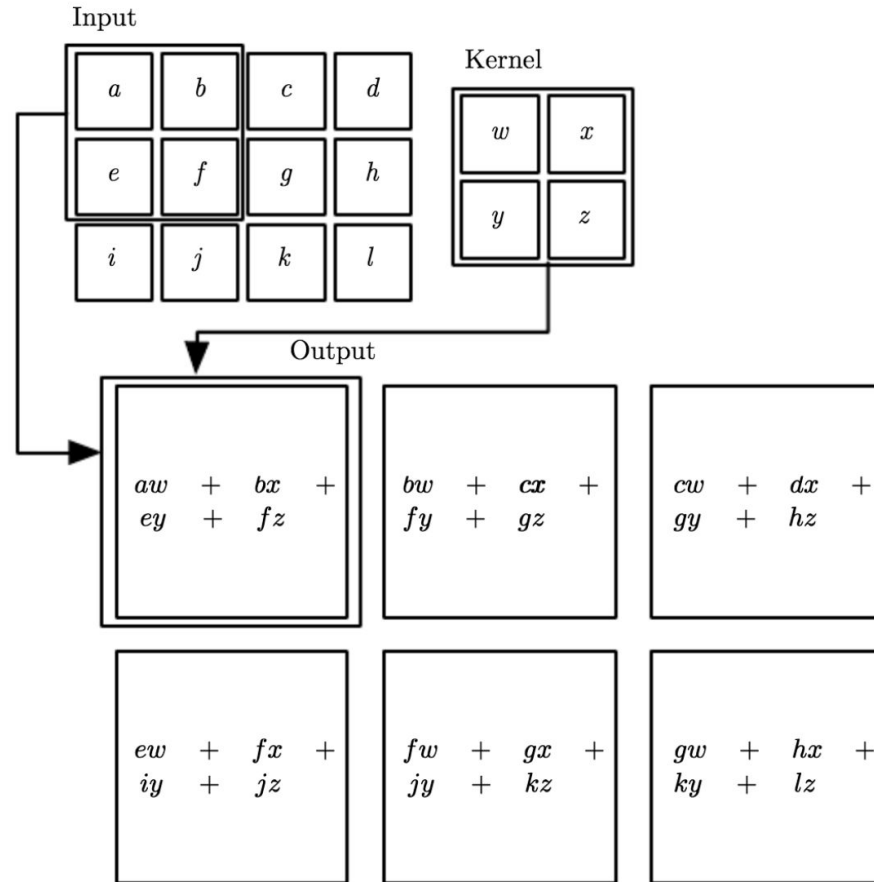


Convolution Layers

- ⦿ The primary purpose of the convolution operation is to extract features from the input
- ⦿ To do this, the input is split into several different areas, and the convolution operation applied to each area
- ⦿ A summary value is then calculated and kept as part of the output
- ⦿ Here, I is the input image (sometimes called **feature map**), K is a two-dimensional **kernel**, or more commonly, **filter**, and is a weighting function
- ⦿ The output resulting from this operation is called the **response map**



The Convolution Operation



The Convolution Operation

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature

Filters

- ◎ The most common filter sizes are 3×3 and 5×5
- ◎ Sometimes 7×7 filters are also used
- ◎ A 1×1 filter is a special case that we will see later in the course when talking about RNNs
- ◎ Odd dimension filters are preferred for computer vision
 - Provide natural padding (we'll see this in the following slides)
 - Ensures a central position or pixel of the filter

Convolution

- ◎ Convolution leverages 3 important ideas that can help improve a machine learning system:
 - **Sparse interactions** / sparse connectivity / sparse weights
 - ◎ Filters are smaller than the input images and thus fewer parameters (weights) need to be stored
 - ◎ This reduces computational expense and improves statistical efficiency
 - **Parameter sharing**
 - ◎ The same parameter is used for more than one function in the model
 - ◎ In a CNN, each element of the filter is applied to every position of the input

Convolution

- **Equivariant representations**

- Parameter sharing causes equivariance to translation: if the input changes, the output changes in the same way
- A function $f(x)$ is invariant to a function g if $f(g(x)) = g(f(x))$
- When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input
- Note that convolution is not equivariant to some other transformations, such as changes in the scale or rotation of an image

◎ **Translation invariant:** After learning a certain pattern from one part of an image, it can recognize it anywhere

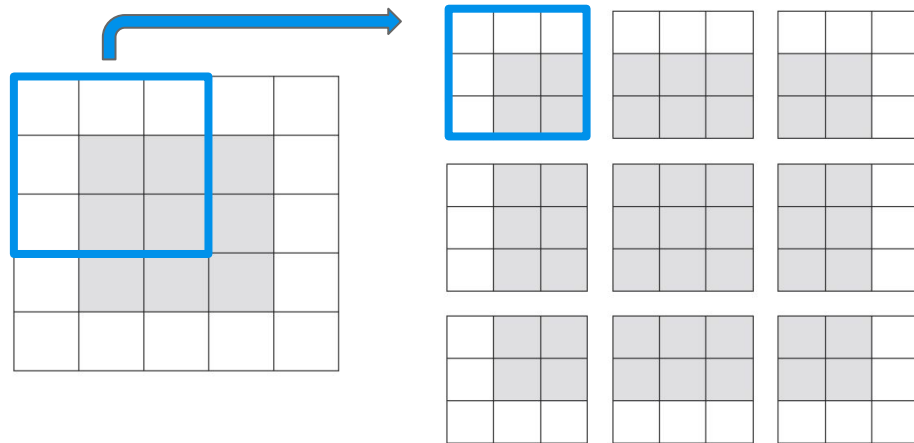
Convolution also provides a way to work with inputs of different size

Padding

- ⦿ Applying a filter to an input image shrinks it - the output dimensions are smaller than the input dimensions
- ⦿ Additionally, when we perform the convolution operation on an input, the pixels on the edges aren't used as much as the pixels in the middle
- ⦿ To make the amount of information used more equal across pixels, you can use **padding**
- ⦿ Padding consists of adding an appropriate number of rows and columns to each side of the image (think a border of pixels)
- ⦿ **This enables an output with the same dimensions as the input**

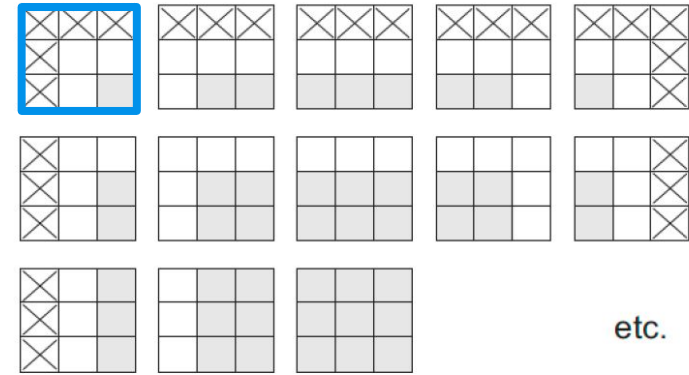
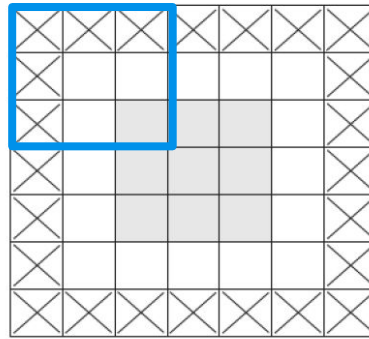
Valid Padding

- In Keras, the default is no padding, or **“valid” padding**
- This means the output will not be the same dimension as the input, and will instead depend on the dimension of the input and the size of the filter
- On the right is a 5 x 5 image
- If we apply a 3 x 3 filter, the output will also be 3 x 3



Same Padding

- On the right is the same 5 x 5 image, but with an added border
- If we use a 3 x 3 filter, we need to add $p = 1$ padding - here, p is the number of rows to add to the border of an image
- The output will then be 5 x 5
- Because the input and output have the same dimensions, this is called **“same” padding**



General Padding Formula

- ◎ There is a general formula that can help you decide how much padding you need or want
 - Let the input be $n \times m$ and the filter $f \times f$
 - Without padding, the output would be:

$$(n - f + 1) \times (m - f + 1)$$

- ◎ For an output with the same dimension as the input, need p such that:

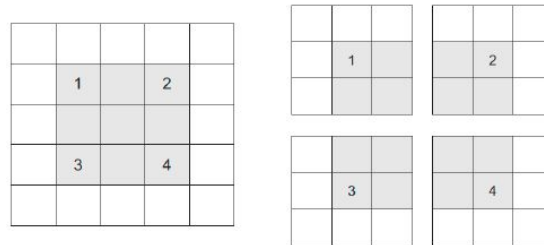
$$n \times m = (n + 2p - f + 1) \times (m + 2p - f + 1)$$

which boils down to $p = \frac{f - 1}{2}$

Convolutional Strides

- Another factor that can influence output size is convolution **strides**
- So far we have assumed that we slide the filter over a single space to extract a new patch - but what if we wanted to slide over 2 spaces, 3 spaces, or more?
- Convolutional strides are convolutions with a stride greater than 1
- If your stride is set equal to 2, you will downsample the width and height of the input image by a factor of 2
- If s is the size of the stride, the output will have dimensions

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{m + 2p - f}{s} + 1 \right\rfloor$$



Pooling

- ◎ Strided convolutions are rarely used in practice - **max-pooling** is more common
- ◎ A typical layer of a CNN consists of 3 stages:
 - **Convolution stage** - linear transformation of the input
 - **Detector stage** - Nonlinear activation (ex: relu activation function is applied)
 - **Pooling stage** - further modification (downsizing) of the output
- ◎ A pooling function replaces the output at a certain location with a summary statistic of the nearby outputs
- ◎ Pooling greatly reduces the computational expense of the network by decreasing the number of parameters to be learned

Pooling

- ⊙ There are different types of pooling

- **Max Pooling**

- ⊙ Outputs the maximum value from a patch for each channel
 - ⊙ Similar to convolution, but instead of transforming patches via a learned linear transformation, they're transformed via a hardcoded max tensor operation
 - ⊙ Very common
 - ⊙ Usually done with 2 x 2 windows

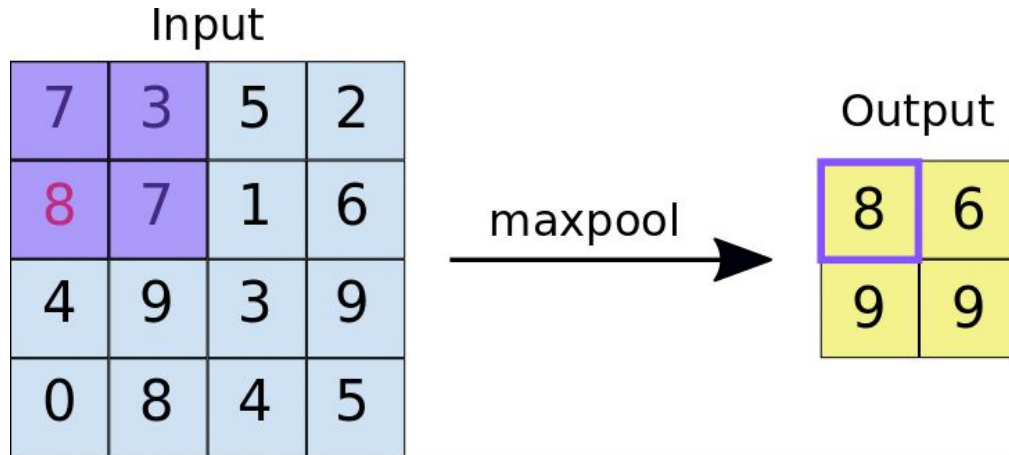
- **Average Pooling**

- **Weighted Pooling**

- ⊙ Based on the distance from the central pixel

Max pooling tends to work better than other pooling functions and convolutional strides

Max Pooling



Average Pooling

Average Pooling (kernel: (2, 2), stride: (2, 2), padding: (0, 0))

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

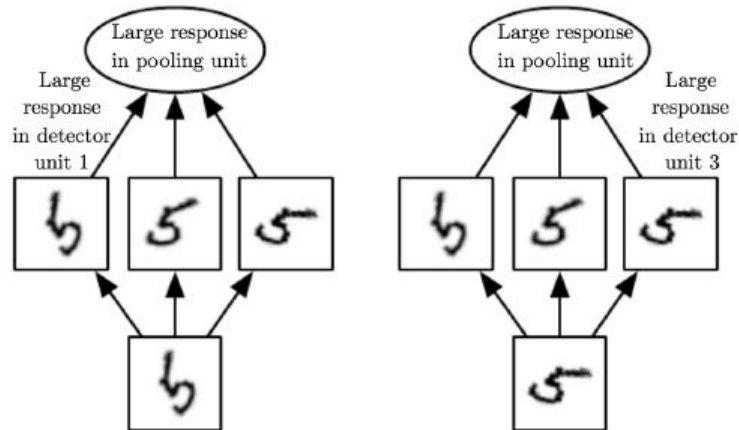
Input

7.75		

Output

Pooling and Invariance

- ⊙ A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input
- ⊙ Here, a set of 3 learned filters and a max pooling unit can learn to become invariant to rotation



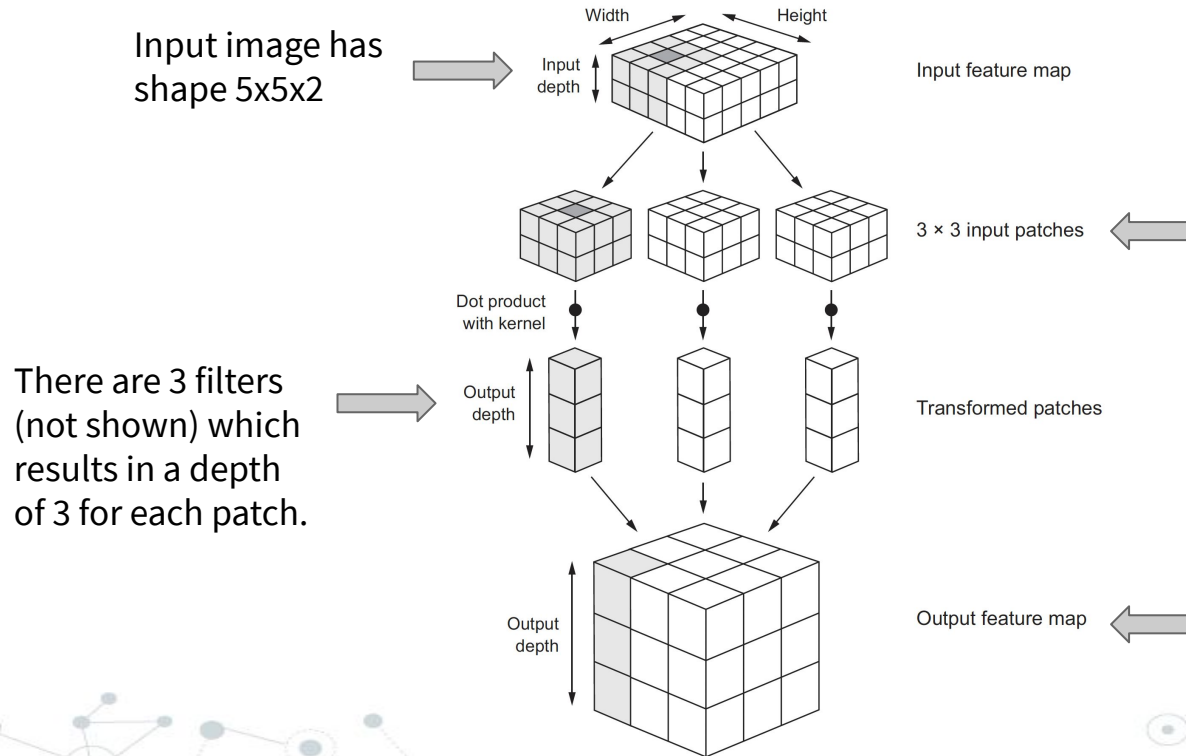
Terminology

- ◎ Convolutions operate over 3D tensors with two spatial axes, **height** and **width**, as well as a **depth** axis (or **channels** axis)
- ◎ For a color (RGB) image, the depth is equal to 3
- ◎ For black and white (grayscale) images, the depth is equal to 1
- ◎ The convolution operation extracts different **patches** from the input image and applies the same transformation to each of them, resulting in a response map that is also a 3D tensor

Terminology

- ◎ The response map has a width, height, and depth, all of which depend on the input image, filter, padding and stride
- ◎ Convolutions are defined by 2 key parameters:
 - Size of the filter
 - Depth of the output response map, i.e., how many filters are applied to the input
- ◎ Convolution works by sliding the filter over the 3D input image, stopping at every possible location, and extracting the 3D patch at each location
 - Each 3D patch is transformed into a 1D vector
 - All 1D vectors are then reassembled into a 3D output map

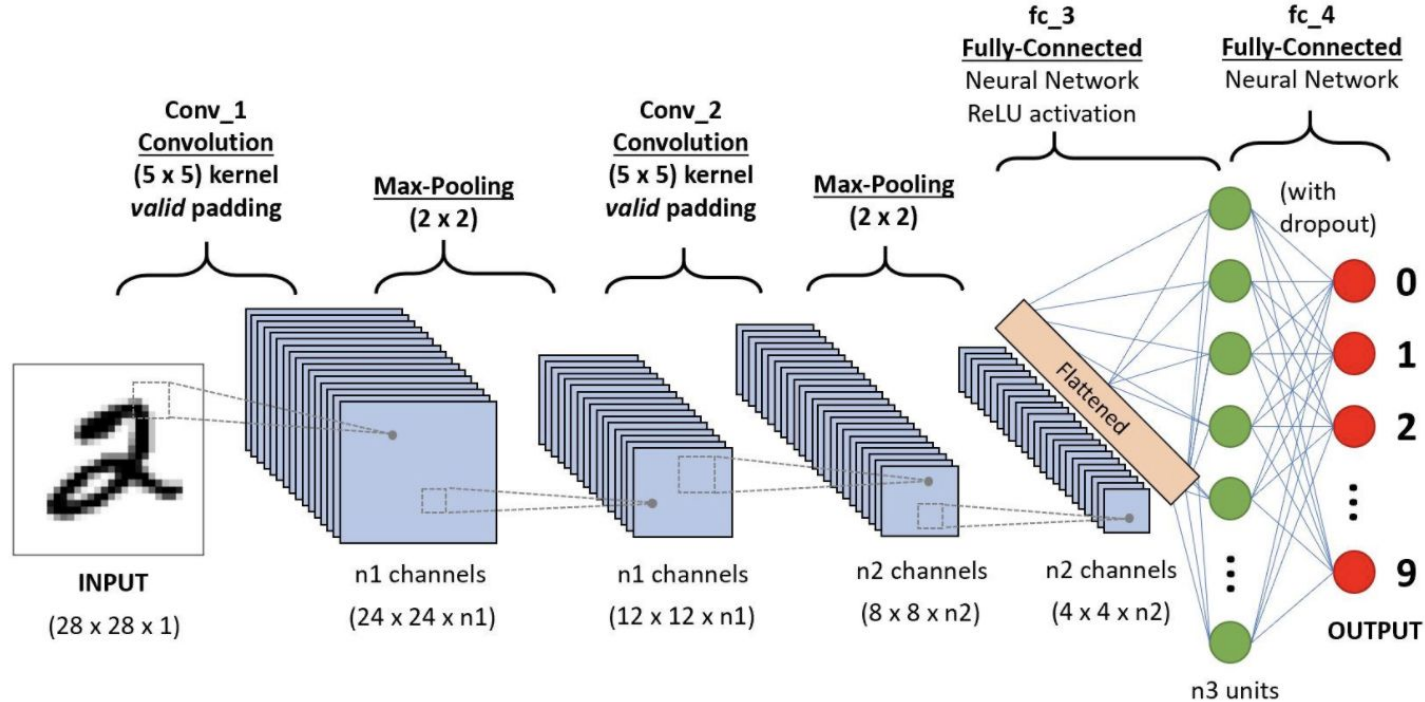
Convolution Schematic



Here, each filter has a shape of $3 \times 3 \times 2$, which results in a total of 9 patches from the original input. Only 3 of those patches are shown here.

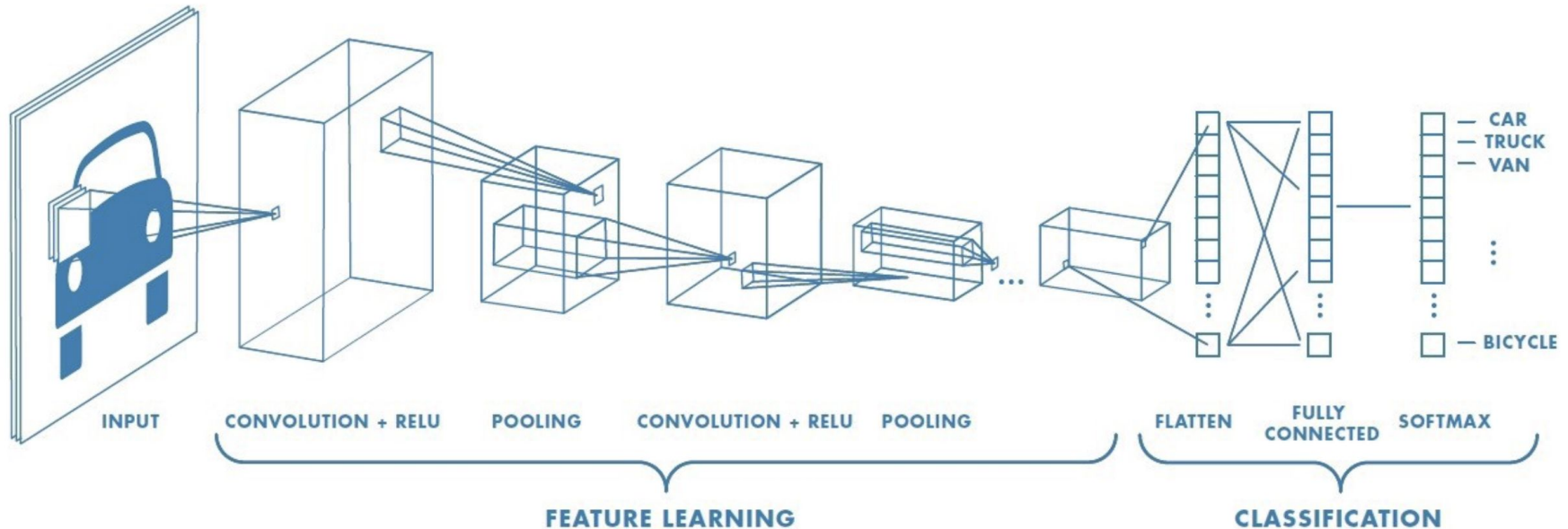
9 total patches, each with a depth of 3 (one number for each filter), resulting in an output shape of $3 \times 3 \times 2$.

CNN Schematic

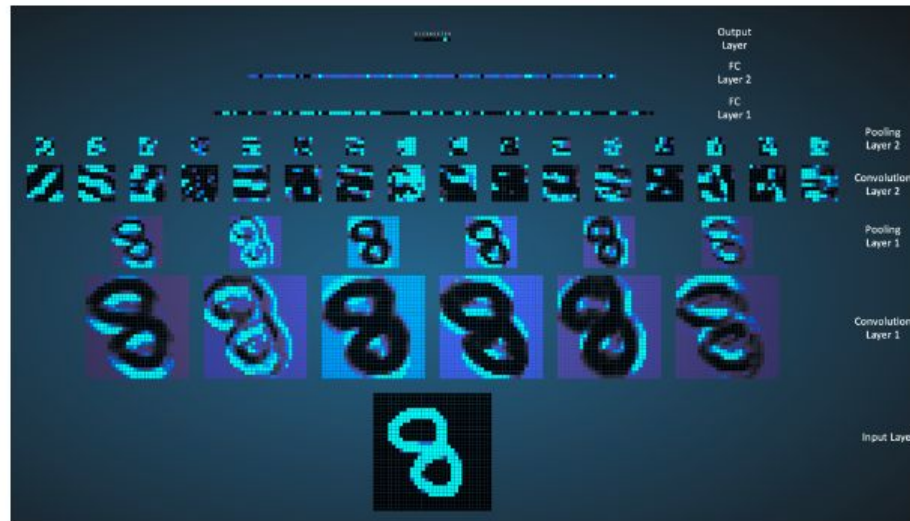


<https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>

CNN Schematic



MNIST Example



MNIST Example

[Colab notebook](#)

MNIST Example

```
1 # Define model
2 model = tf.keras.models.Sequential([
3     # Convolutional layer with 32 filters that are 3x3, relu activation function
4     tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
5     # Pooling layer with 2x2 windows
6     tf.keras.layers.MaxPooling2D((2, 2)),
7
8     # Convolutional layer with 64 filters that are 3x3, relu activation function
9     tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
10    # Pooling layer with 2x2 windows
11    tf.keras.layers.MaxPooling2D((2, 2)),
12
13    # Convolutional layer with 64 filters that are 3x3, relu activation function
14    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
15
16    # Collapse the 3D tensor
17    tf.keras.layers.Flatten(),
18
19    # Fully connected layer with 64 hidden units, relu activation function
20    tf.keras.layers.Dense(64, activation='relu'),
21
22    # Softmax output function with 10 classes
23    tf.keras.layers.Dense(10, activation='softmax')
24 ])
```