Intro to Deep Learning

IAP Computing, Optimization, and Statistics

Jan 13, 2022

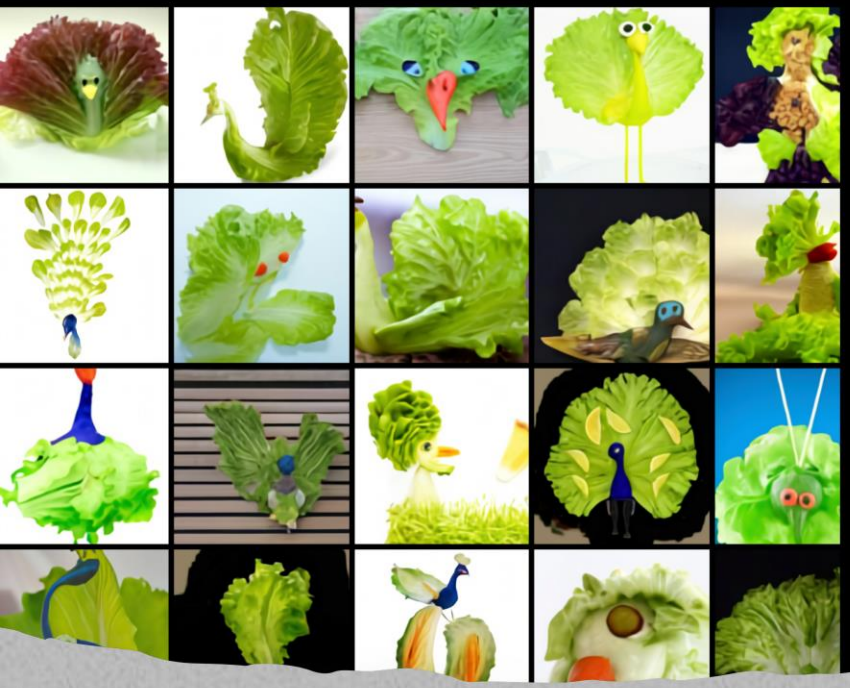
Outline

1. Fundamental concepts on Neural Networks
    1. Intro/Motivation
    2. Perceptron
    3. Neural Network Architecture
    4. SGD and Back-Propagation
2. Tensorflow/Keras Tutorial
3. Convolutional Neural Networks
    1. Convolutional layer
    2. Pooling Layer
4. Image Classification using CNN
5. If time permits
    1. Transfer Learning
    2. Hyperparameter tuning and model selection
    3. Data Augmentation

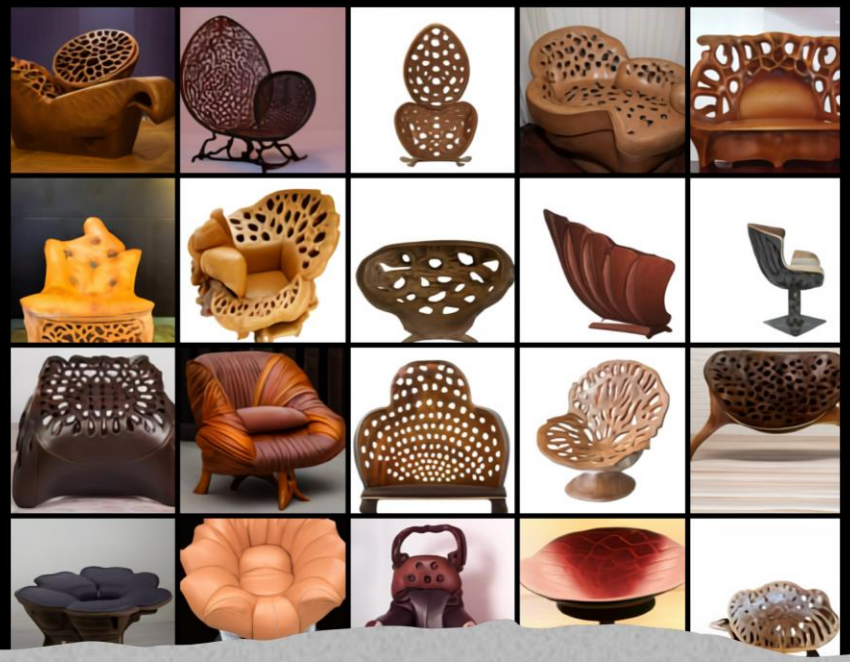
Let’s start with some cool AI stuff

Open AI's GPT-3

a peacock made of lettuce. a peacock with the texture of a lettuce.



a leather armchair in the style of a lotus root. a leather armchair imitating a lotus root.

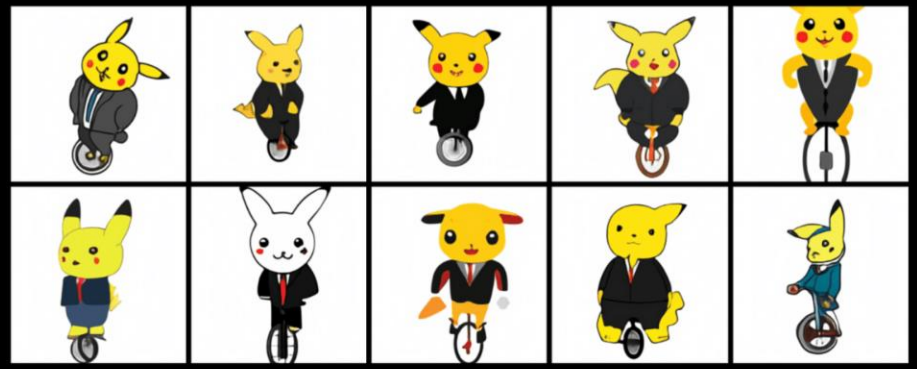


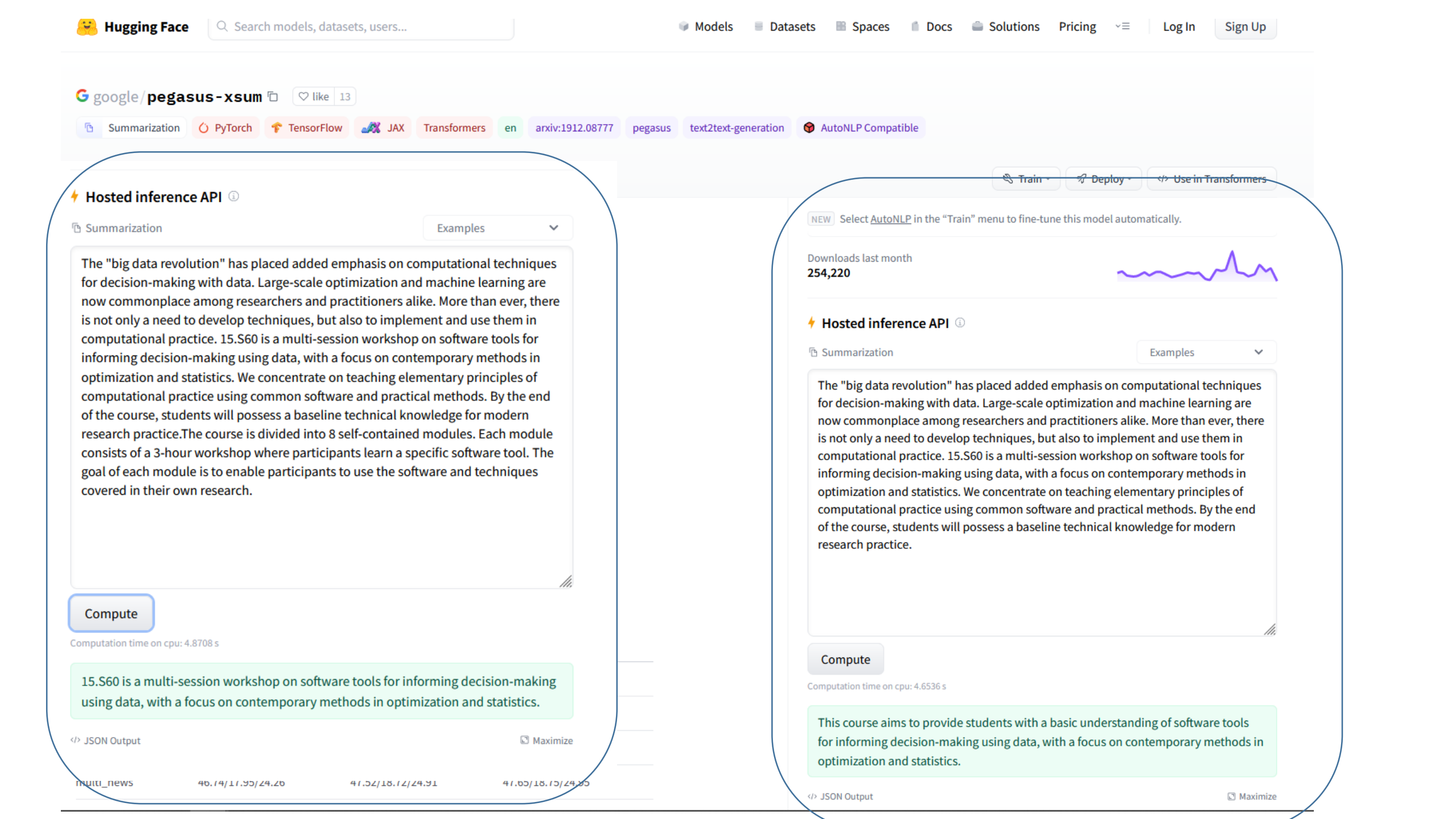
a torus made of rainbow. a torus with the texture of rainbow.

DALL-E

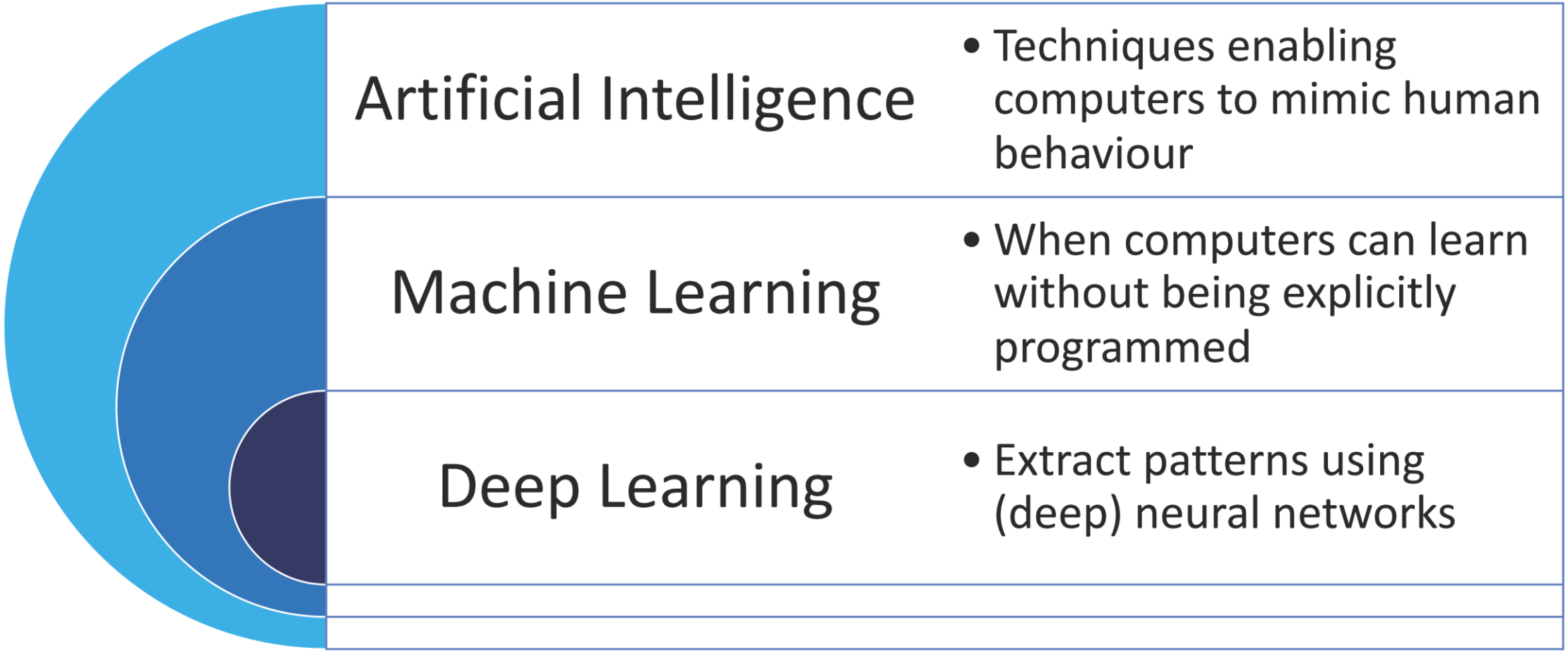
an illustration of a pikachu in a suit riding a unicycle

AI-GENERATED IMAGES




Hugging Face
Search models, datasets, users...
Models
Datasets
Spaces
Docs
Solutions
Pricing
Log In
Sign Up
google / pegasus-xsum
like 13
Summarization
PyTorch
TensorFlow
JAX
Transformers
en
arxiv:1912.08777
pegasus
text2text-generation
AutoNLP Compatible
Train
Deploy
Use in Transformers
Hosted inference API
Summarization
Examples
The "big data revolution" has placed added emphasis on computational techniques for decision-making with data. Large-scale optimization and machine learning are now commonplace among researchers and practitioners alike. More than ever, there is not only a need to develop techniques, but also to implement and use them in computational practice. 15.S60 is a multi-session workshop on software tools for informing decision-making using data, with a focus on contemporary methods in optimization and statistics. We concentrate on teaching elementary principles of computational practice using common software and practical methods. By the end of the course, students will possess a baseline technical knowledge for modern research practice.The course is divided into 8 self-contained modules. Each module consists of a 3-hour workshop where participants learn a specific software tool. The goal of each module is to enable participants to use the software and techniques covered in their own research.
Compute
Computation time on cpu: 4.8708 s
15.S60 is a multi-session workshop on software tools for informing decision-making using data, with a focus on contemporary methods in optimization and statistics.
JSON Output
Maximize
multi_news
46.74/17.95/24.26
47.52/18.72/24.91
47.65/18.75/24.95
NEW Select AutoNLP in the "Train" menu to fine-tune this model automatically.
Downloads last month
254,220
Hosted inference API
Summarization
Examples
The "big data revolution" has placed added emphasis on computational techniques for decision-making with data. Large-scale optimization and machine learning are now commonplace among researchers and practitioners alike. More than ever, there is not only a need to develop techniques, but also to implement and use them in computational practice. 15.S60 is a multi-session workshop on software tools for informing decision-making using data, with a focus on contemporary methods in optimization and statistics. We concentrate on teaching elementary principles of computational practice using common software and practical methods. By the end of the course, students will possess a baseline technical knowledge for modern research practice.
Compute
Computation time on cpu: 4.6536 s
This course aims to provide students with a basic understanding of software tools for informing decision-making using data, with a focus on contemporary methods in optimization and statistics.
JSON Output
Maximize

What is Deep Learning?



What is the history behind the current AI boom?

1952 Stochastic Gradient Descent

1958 Perceptron

Learnable weights

1986 Backpropagation

Multi-Layer Perceptron

1995 Deep Convolutional NN

Digit Recognition

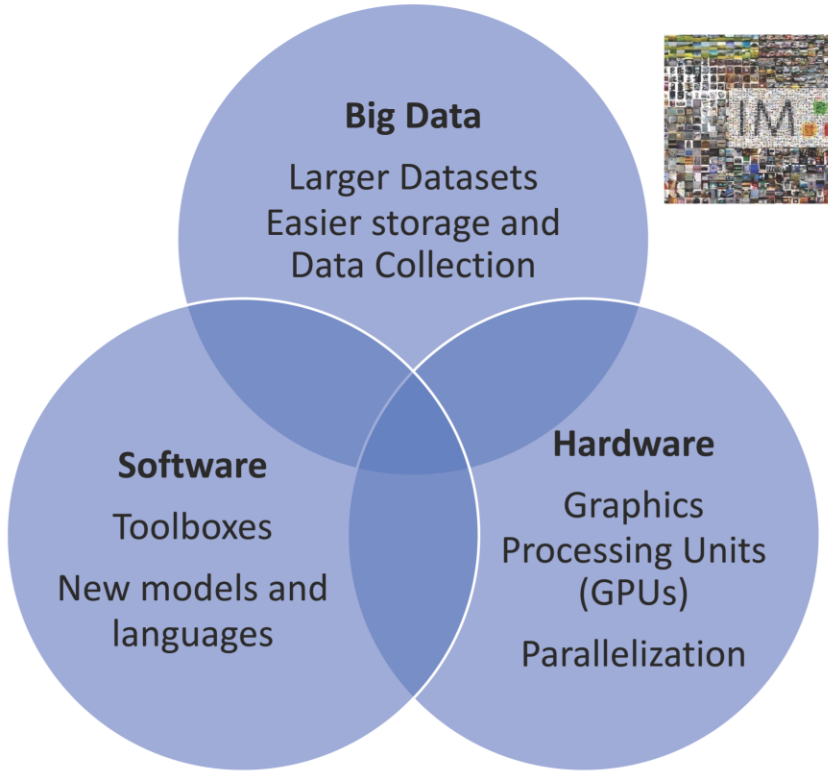
2012 ImageNet competition

2019 Turing Award for Deep Learning

WIKIPEDIA
The Free Encyclopedia

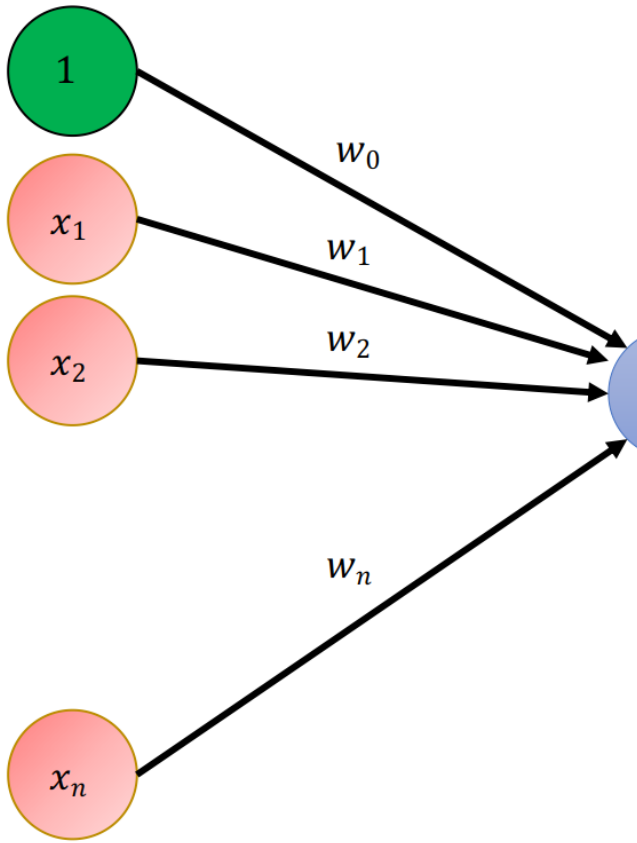
TensorFlow

PyTorch



Neural Networks Basics

The Perceptron



Input $x = (x_1, \ldots, x_n)$

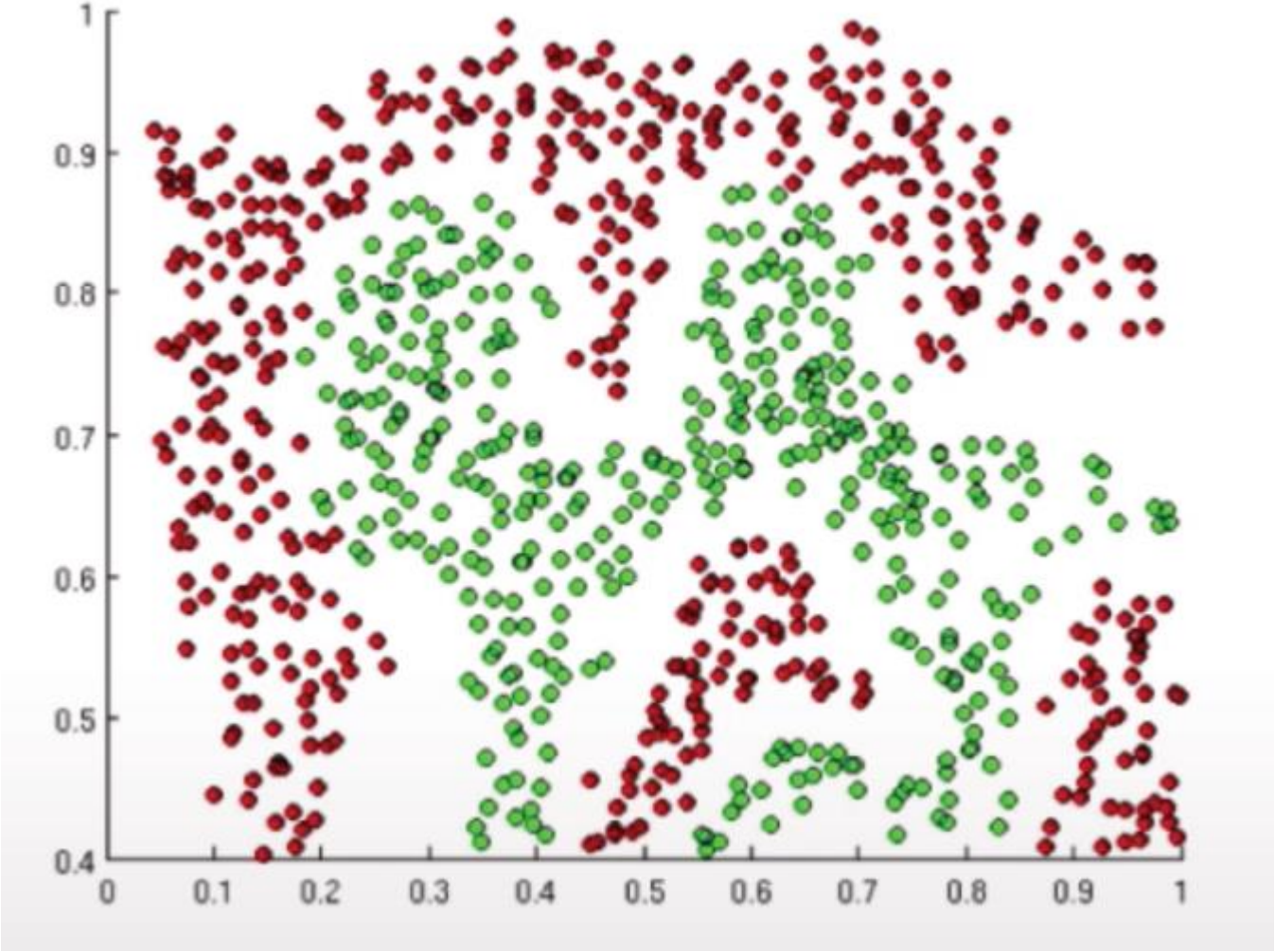
$g : \mathbb{R} \to \mathbb{R}$

$y = g(z)$

Output

$$z = w_0 + \sum_i w_i x_i$$

Classification task

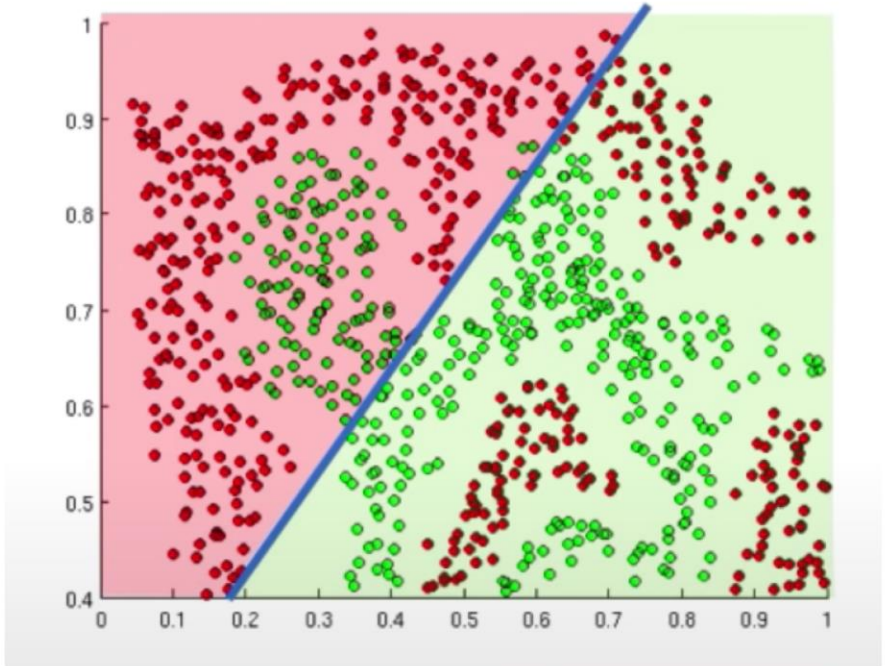


Activation functions

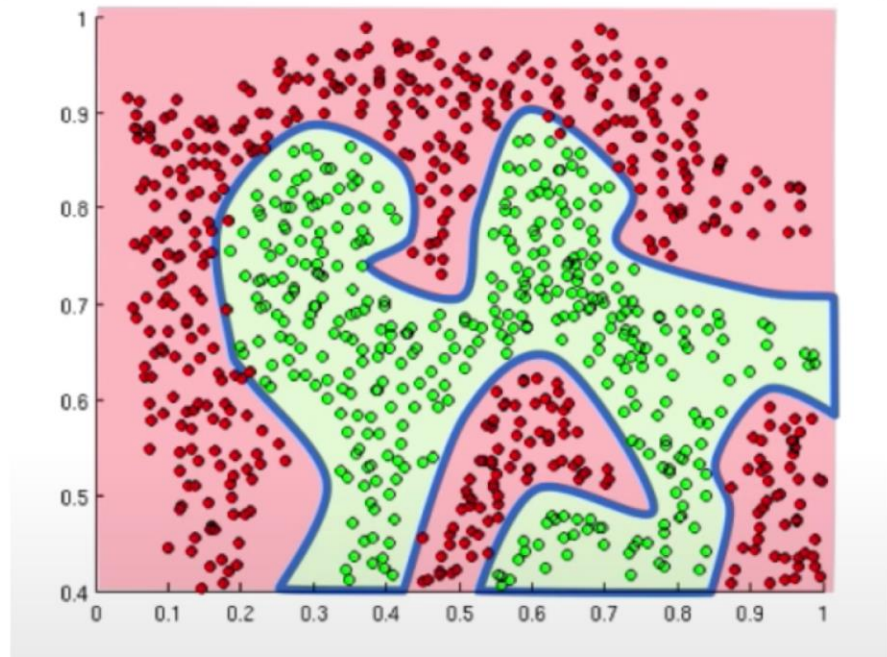
$$z = w_0 + \sum_i w_i x_i$$

$$y = g(z)$$

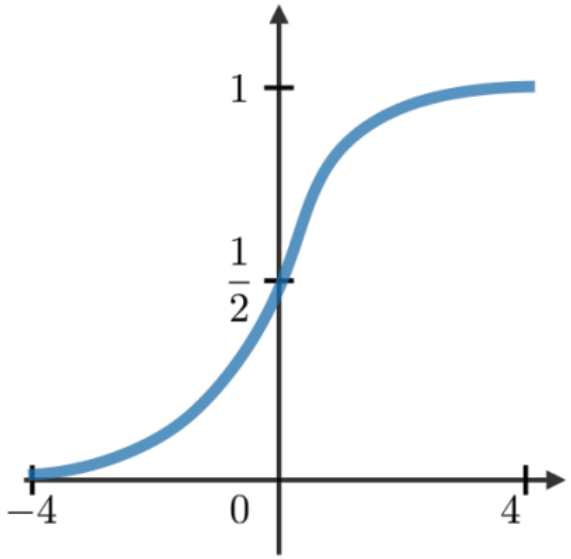
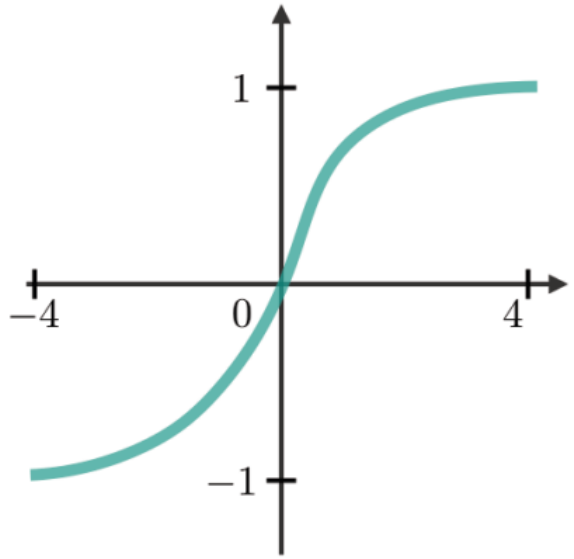
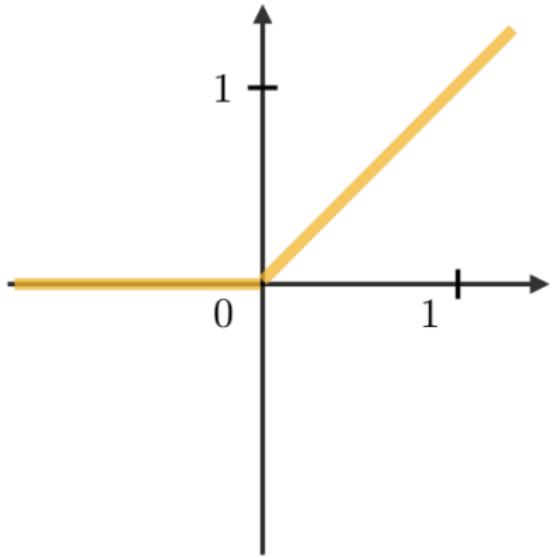
Linear activation functions produce linear decisions no matter the network size



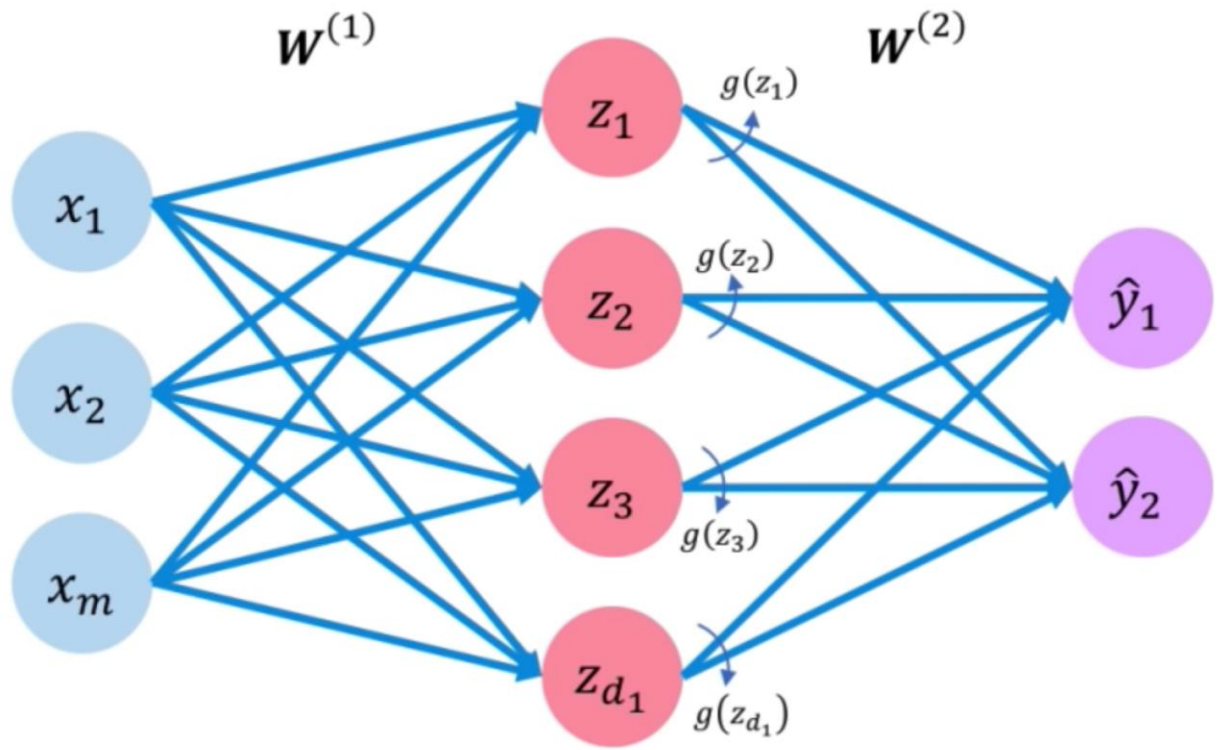
Non-linear activation can help approximate arbitrarily complex functions



Activation Functions

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$
		

One-layer Neural Network

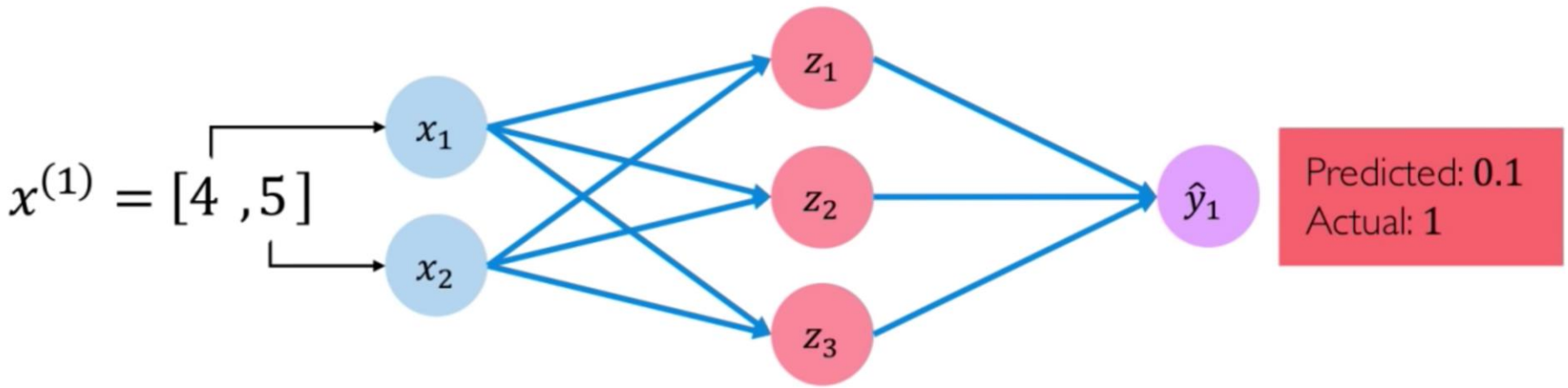


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

Picture source: http://introtodeeplearning.com/

Loss function

The loss of our network measures the cost incurred from incorrect predictions.



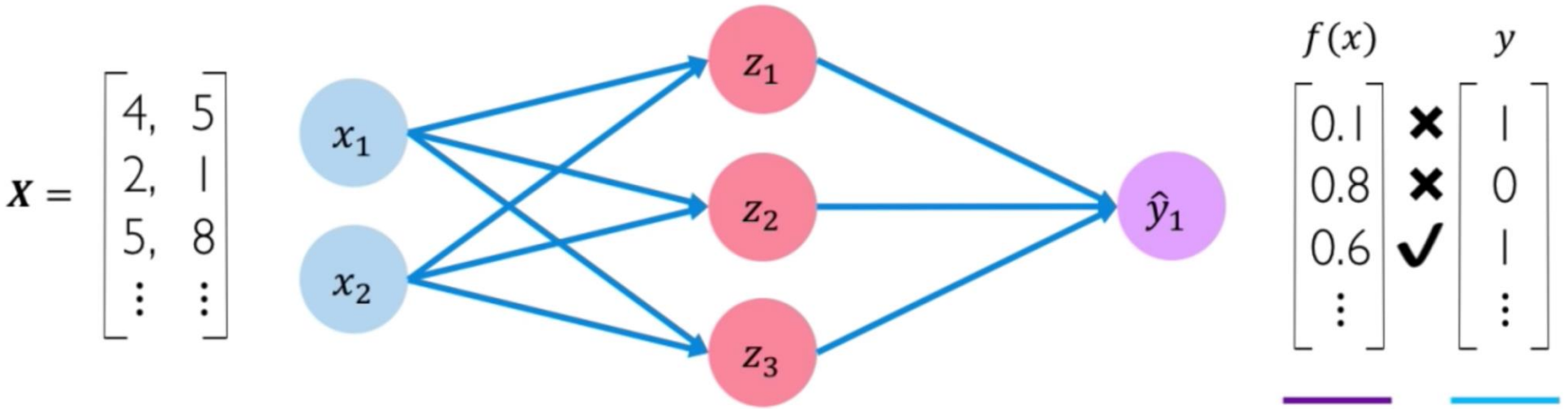
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Picture source: http://introtodeeplearning.com/

Empirical Loss

The **empirical loss** measures the total loss over our entire dataset.



Also known as:
- Objective function
- Cost function
- Empirical Risk

$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

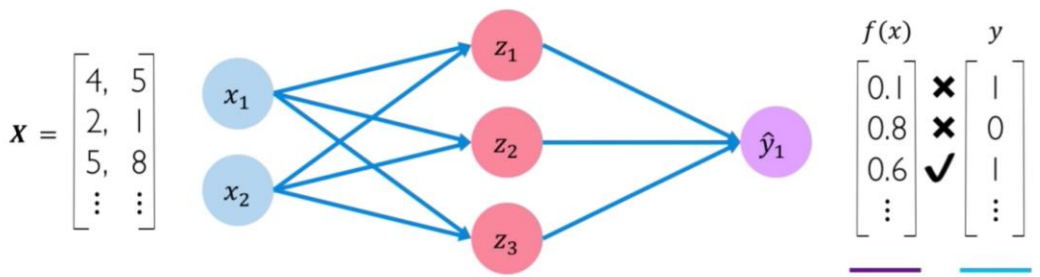
Predicted Actual

Picture source: http://introtodeeplearning.com/

Different loss functions

Binary Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*



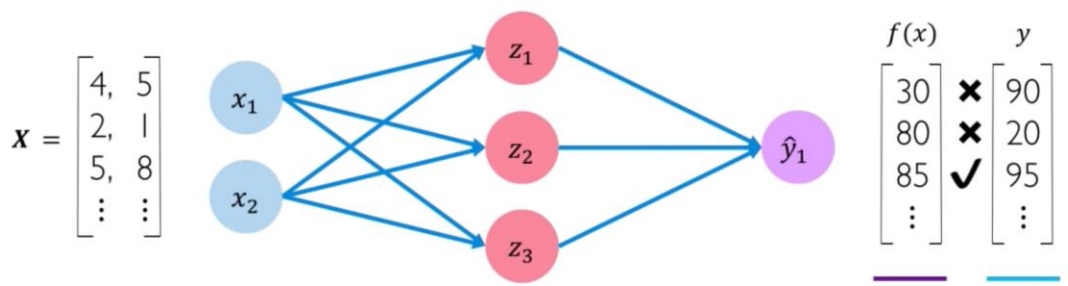
$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)$$

Actual Predicted Actual Predicted

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
```

Picture source: http://introtodeeplearning.com/

How to optimize the loss function?

We need to find the network weights leading to the lowest loss.

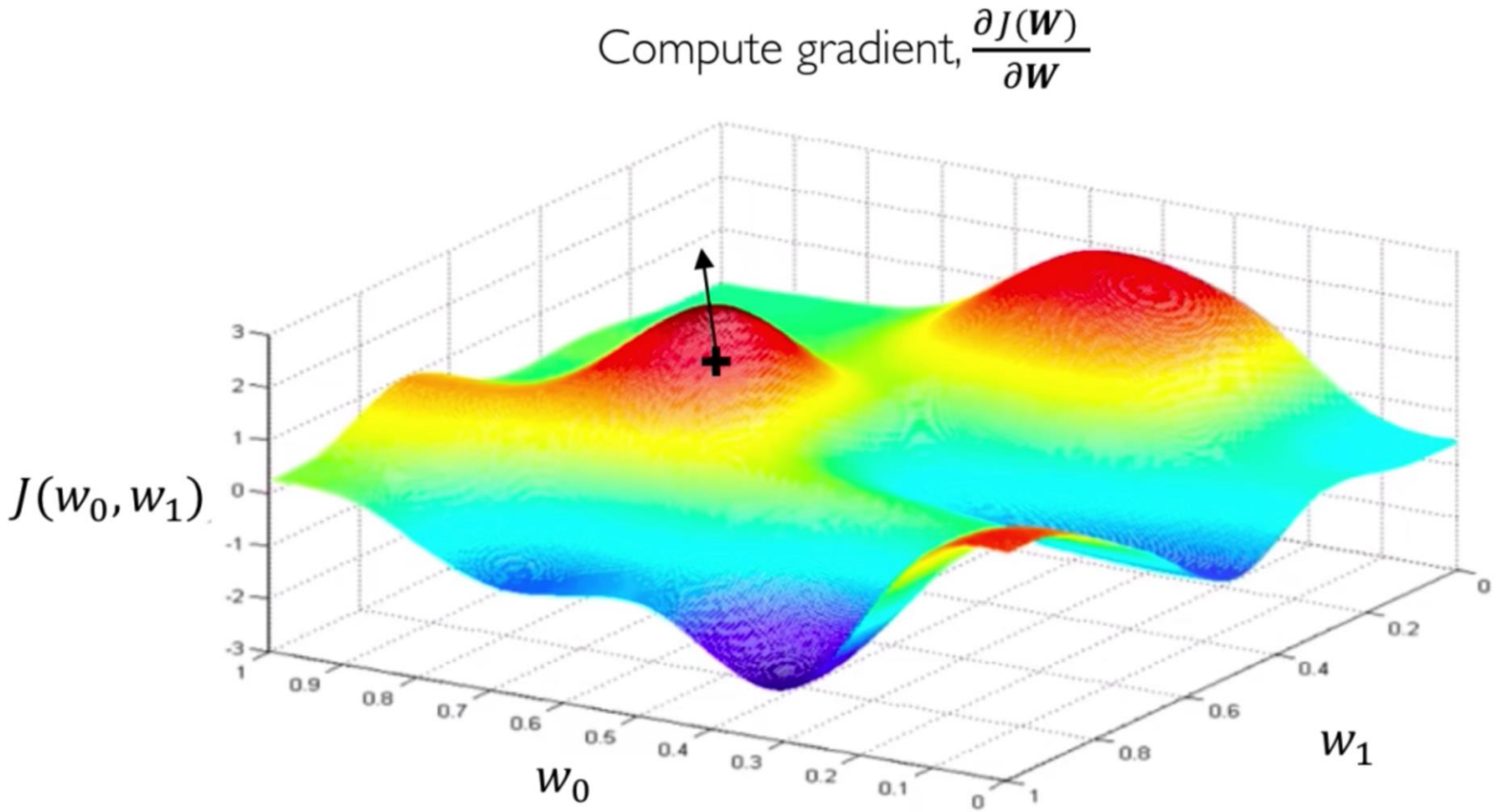
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:

$$\boldsymbol{W} = \left\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\right\}$$

But it looks like this...



Picture source: http://introtodeeplearning.com/

Fortunately there is Gradient Descent!

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

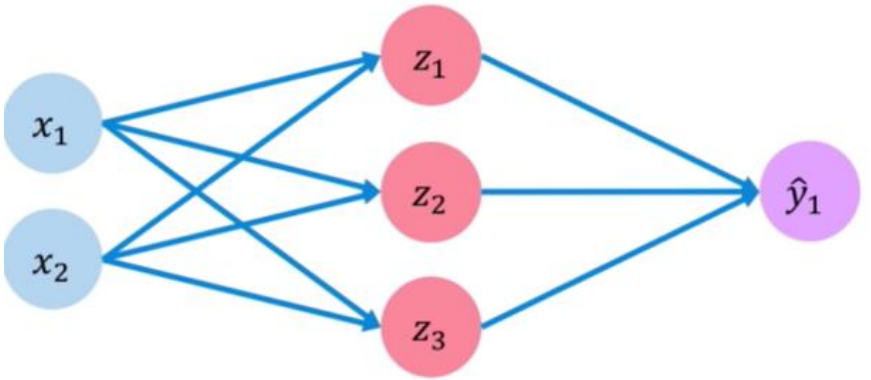
Next problem: compute gradients

We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(2)})}{\partial W} = \frac{1}{n} \frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(2)}}$$

Chain rule!



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(x^{(i)}; W), y^{(i)}\right)$$

$= \hat{y}$

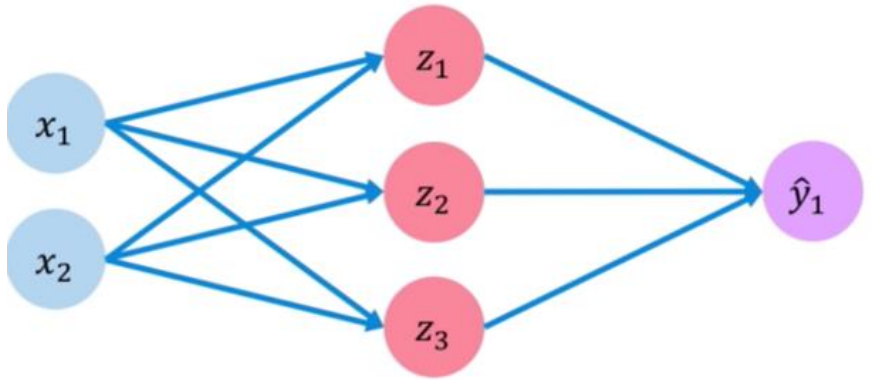
Next problem: compute gradients

We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(2)})}{\partial W} = \frac{1}{n} \frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(2)}}$$

$$= \frac{1}{n} \frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z) z_{ji}$$



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \quad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f(x^{(i)}; W), y^{(i)}\right)$$

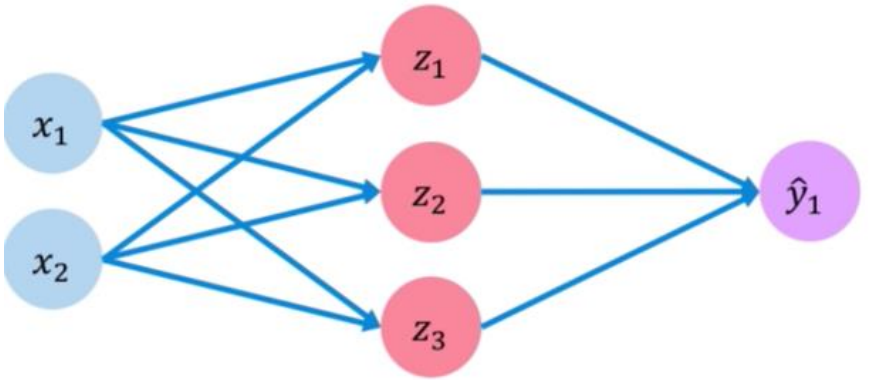
$= \hat{y}$

Next problem: compute gradients

We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(1)})}{\partial W} = \frac{1}{n} \frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(1)}}$$



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \, w_{j,i}^{(2)}\right)$$

$$\boldsymbol{J(W)} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\big(\underbrace{f\big(x^{(i)}; \boldsymbol{W}\big)}_{=\hat{y}}, \underline{y^{(i)}}\big)$$

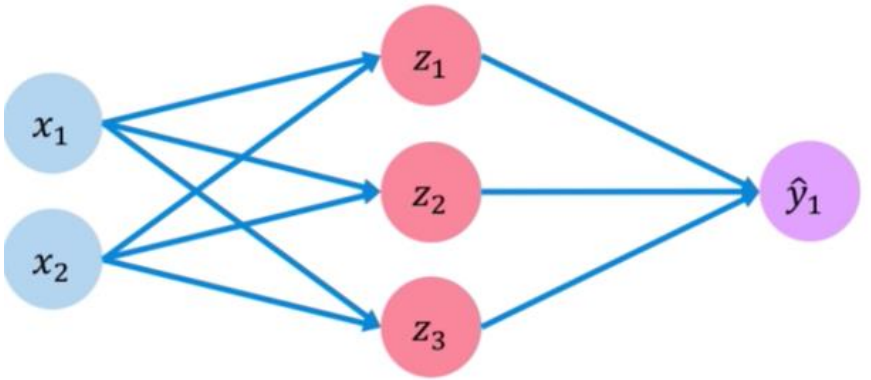
Next problem: compute gradients

We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(1)})}{\partial W} = \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(1)}}$$

$$= \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z)\frac{\partial (w^{(2)\intercal} z)}{\partial w_{ji}^{(1)}}$$



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \quad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f(x^{(i)}; W), y^{(i)}\right)$$

$= \hat{y}$

Next problem: compute gradients

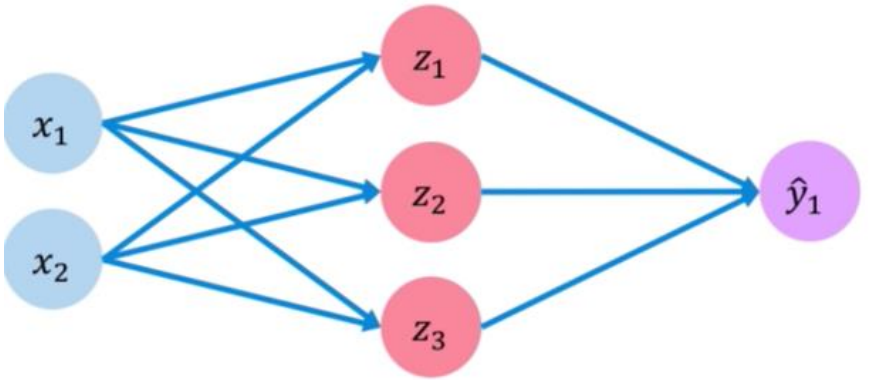
We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(1)})}{\partial W} = \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(1)}}$$

$$= \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z)\frac{\partial (w^{(2)\intercal} z)}{\partial w_{ji}^{(1)}}$$

$$= \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z) w_{ji}^{(2)} \frac{\partial z_j}{\partial w_{ji}^{(1)}}$$



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \quad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f(x^{(i)}; W), y^{(i)}\right)$$

$= \hat{y}$

Next problem: compute gradients

We need to compute the gradient $\frac{\partial \mathcal{J}(W)}{\partial W}$ with respect to all parameters $W = (w_{ij}^{(l)})_{\{l;i,j\}}$.

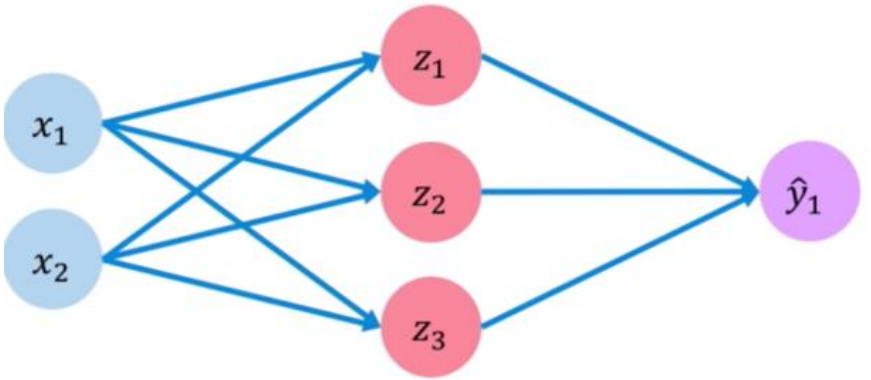
Let's begin with a neural network with one layer:

$$\frac{\partial \mathcal{J}(w_{ji}^{(1)})}{\partial W} = \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y}, y_i)}{\partial \hat{y}} \cdot \frac{\partial g}{\partial w_{ji}^{(1)}}$$

$$= \frac{1}{n}\frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z)\frac{\partial (w^{(2)\intercal} z)}{\partial w_{ji}^{(1)}}$$

$$= \underbrace{\frac{1}{n}\frac{\partial \mathcal{L}(\hat{y})}{\partial \hat{y}} \cdot g'(w^{(2)\intercal} z) w_{ji}^{(2)}}\frac{\partial z_j}{\partial w_{ji}^{(1)}}$$

We already computed this!

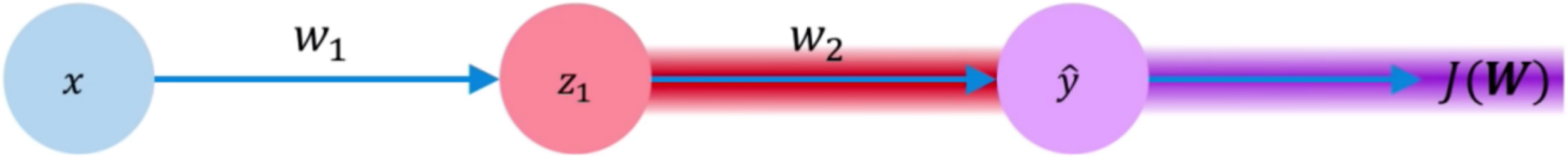


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)} \quad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\big(f(x^{(i)}; W), y^{(i)}\big)$$

$= \hat{y}$

Backpropagation (aka chain rule)



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

But how do you choose the learning rate?

Remember:

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

How can we set the learning rate?

Ideas?

Adaptative Learning Rate

Learning rate is not fixed anymore, instead:

You can adapt it depending on:
- How large gradients are
- How fast the learning is happening (momentum)
- ...

In Deep Learning, Adam is an extremely popular method and does all this automatically. But you still need to choose a base value (typically between $10^{-3}/5.10^{-5}$ but depends on the task and architecture.)

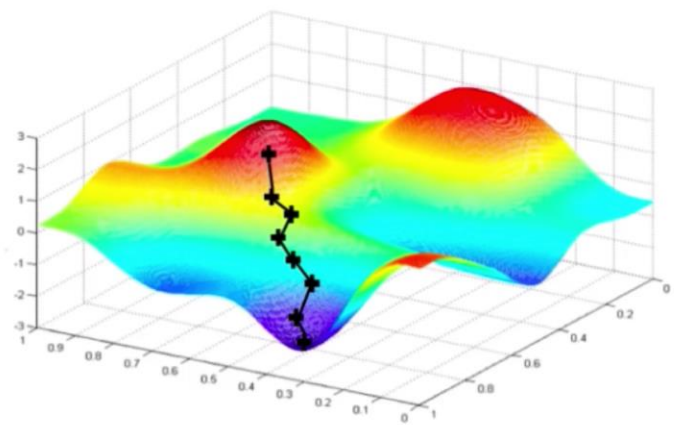
Another problem: computing gradients is expensive!

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Can be very **computationally intensive** to compute!



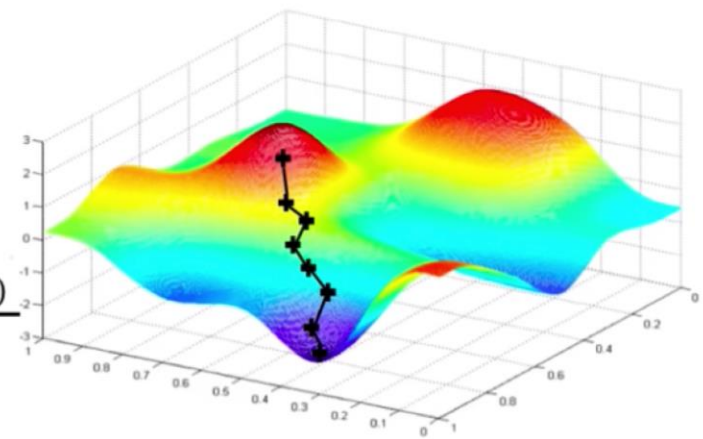
Picture source: http://introtodeeplearning.com/

Another problem: computing gradients is expensive!

Stochastic Gradient Descent

Algorithm

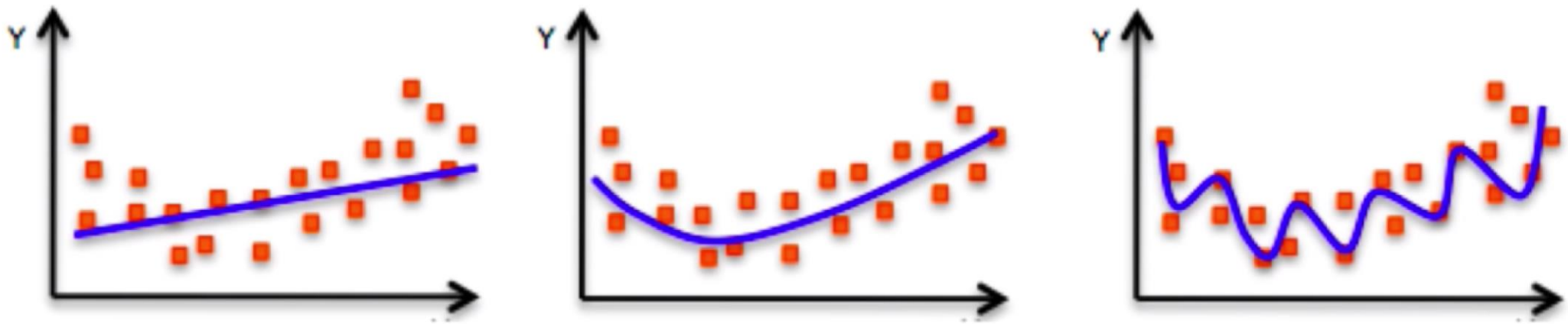
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



We use mini-batches while training allowing for a smoother convergence and larger learning rates.

Picture source: http://introtodeeplearning.com/

Another classic ML problem: overfitting



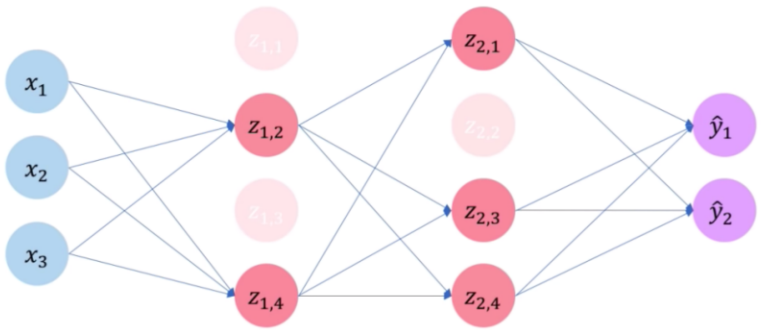
Picture source: http://introtodeeplearning.com/

Let's overcome this! Ideas?

Regularization 1: Dropout

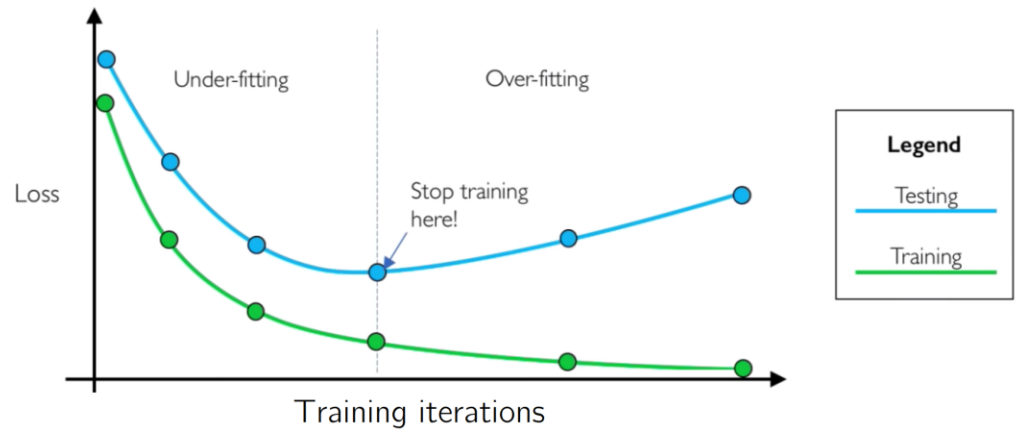
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.keras.layers.Dropout(p=0.5)
```



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



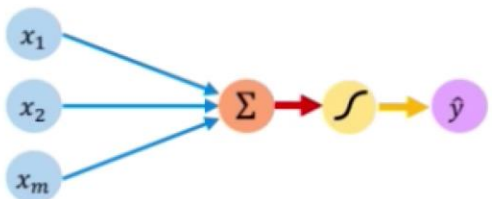
Regularization 3: L1 or L2 penalty on the weights

Picture source: http://introtodeeplearning.com/

Review

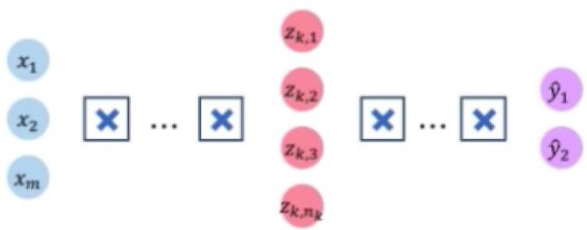
The Perceptron

- Structural building blocks
- Nonlinear activation functions



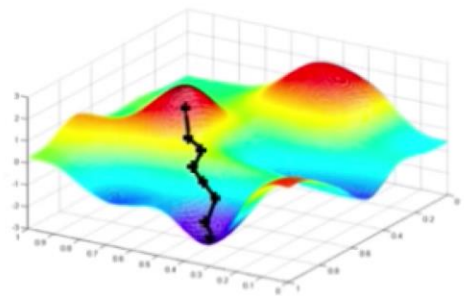
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Picture source: http://introtodeeplearning.com/

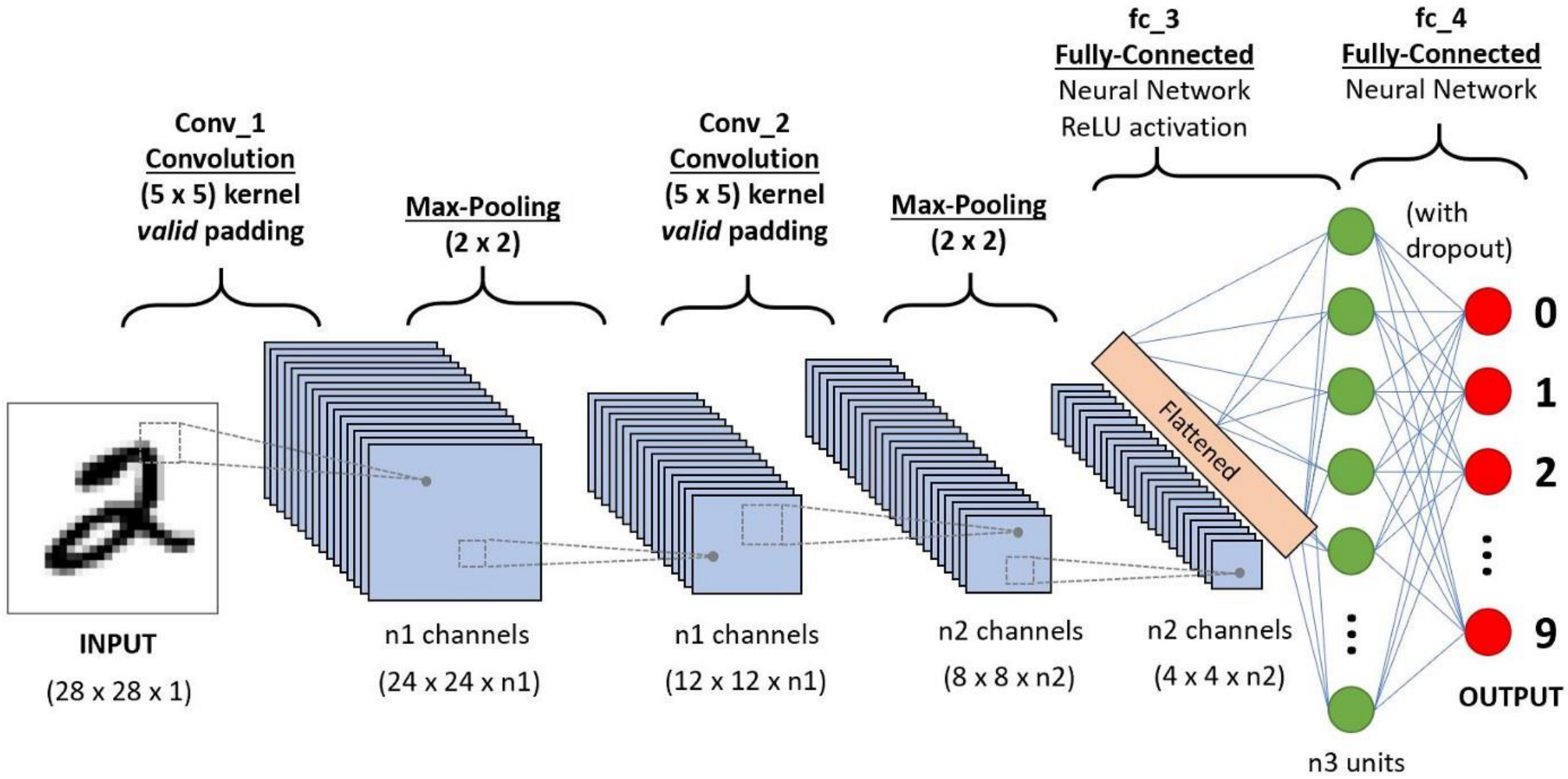
Let’s get hands on

https://colab.research.google.com/drive/1abnkCeDxtwQPqauK7XjxrI6ewZesPjzv?usp=sharing

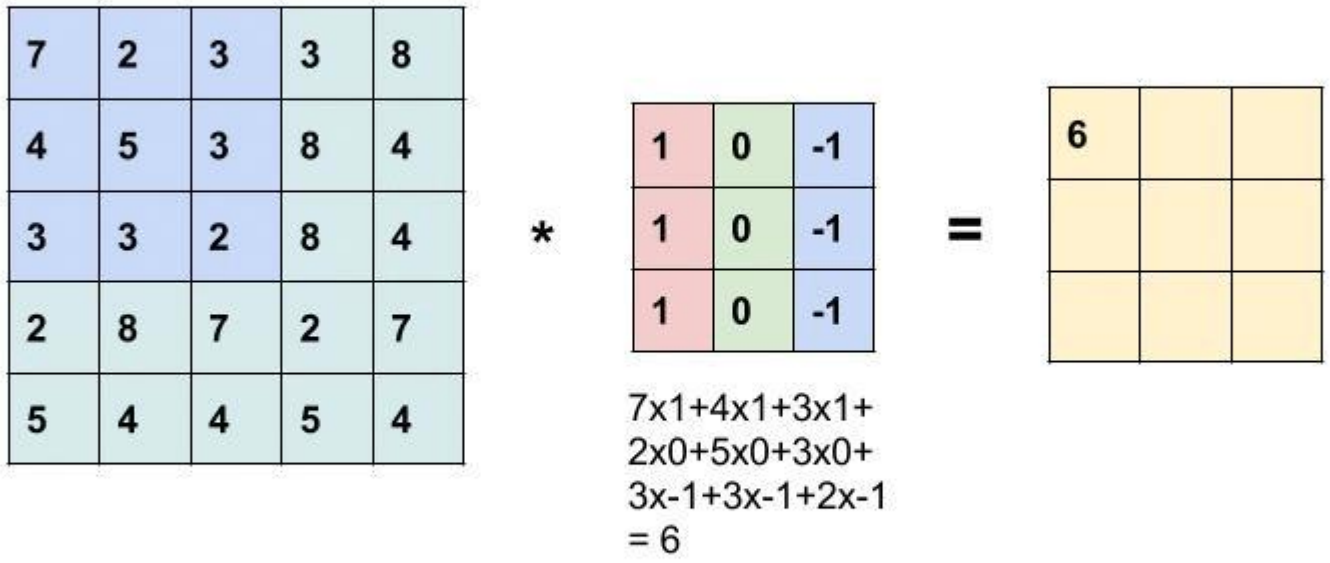
Break

Convolutional Neural Networks

CNN are well suited for images as inputs

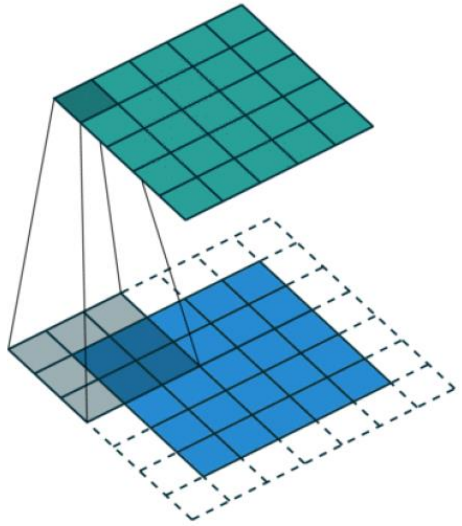


Convolutional Layer

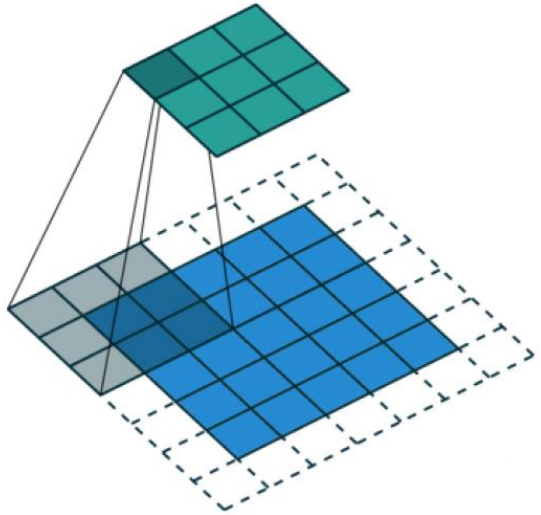


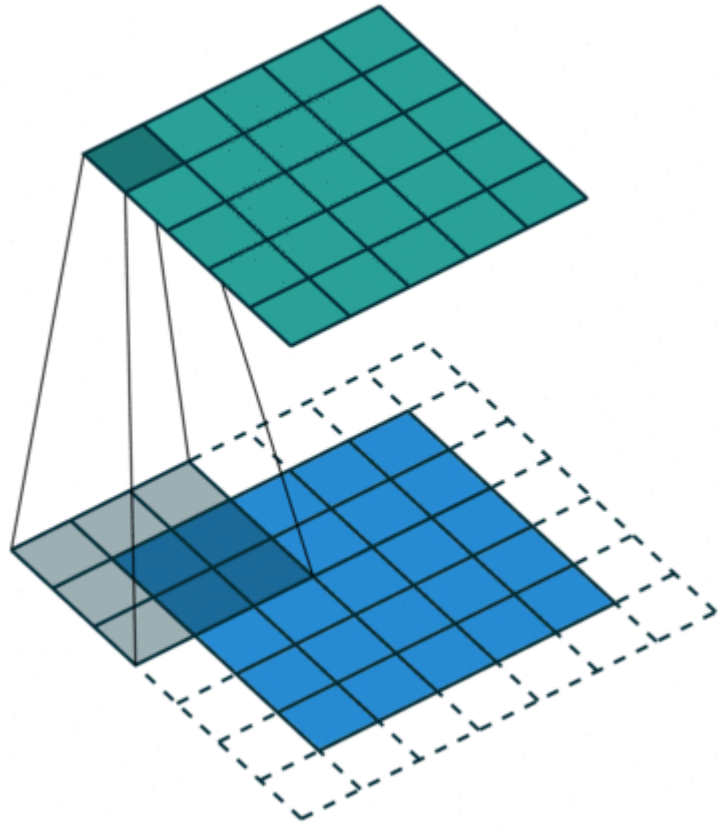
Convolutional Layer

Padding: 1
Stride: 1



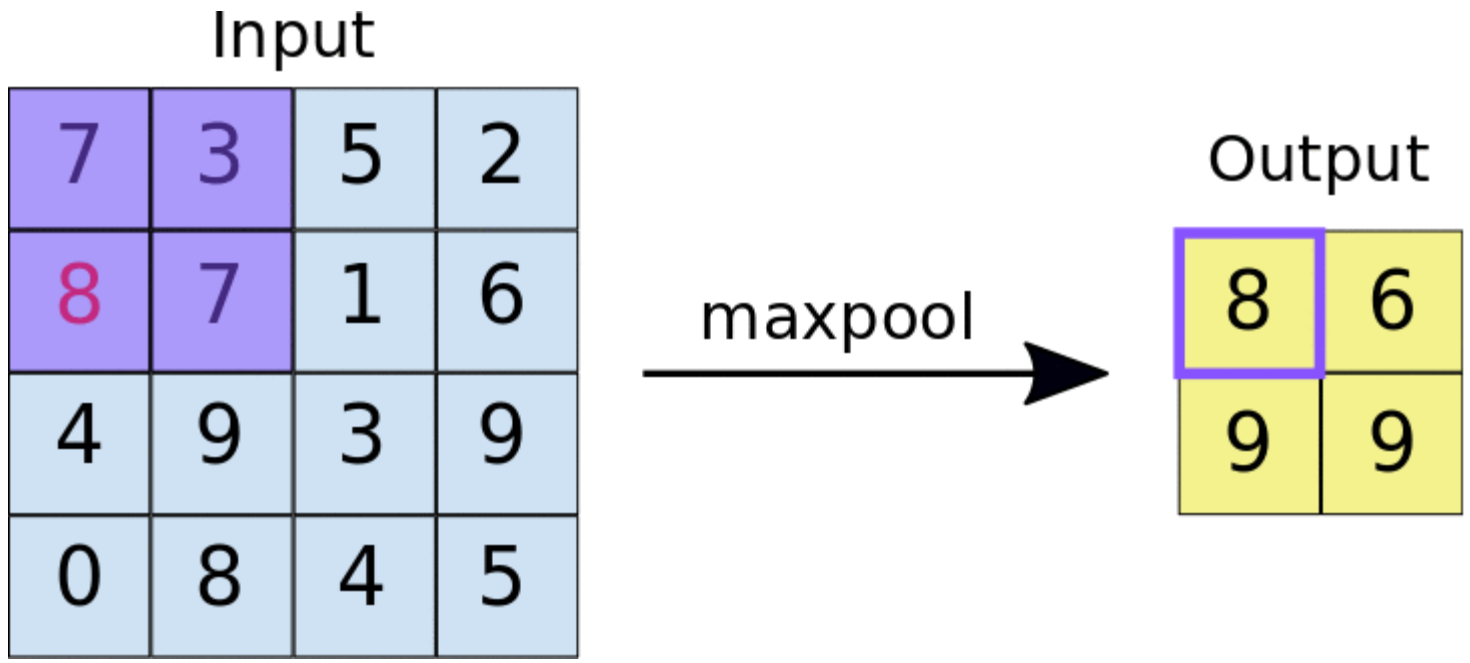
Padding: 1
Stride: 2





Pooling Layer

- Decreases dimensionality.
- Highlights relevant features.
- Some sort of ‘regularization’



Let’s try it in Fashion MNIST!