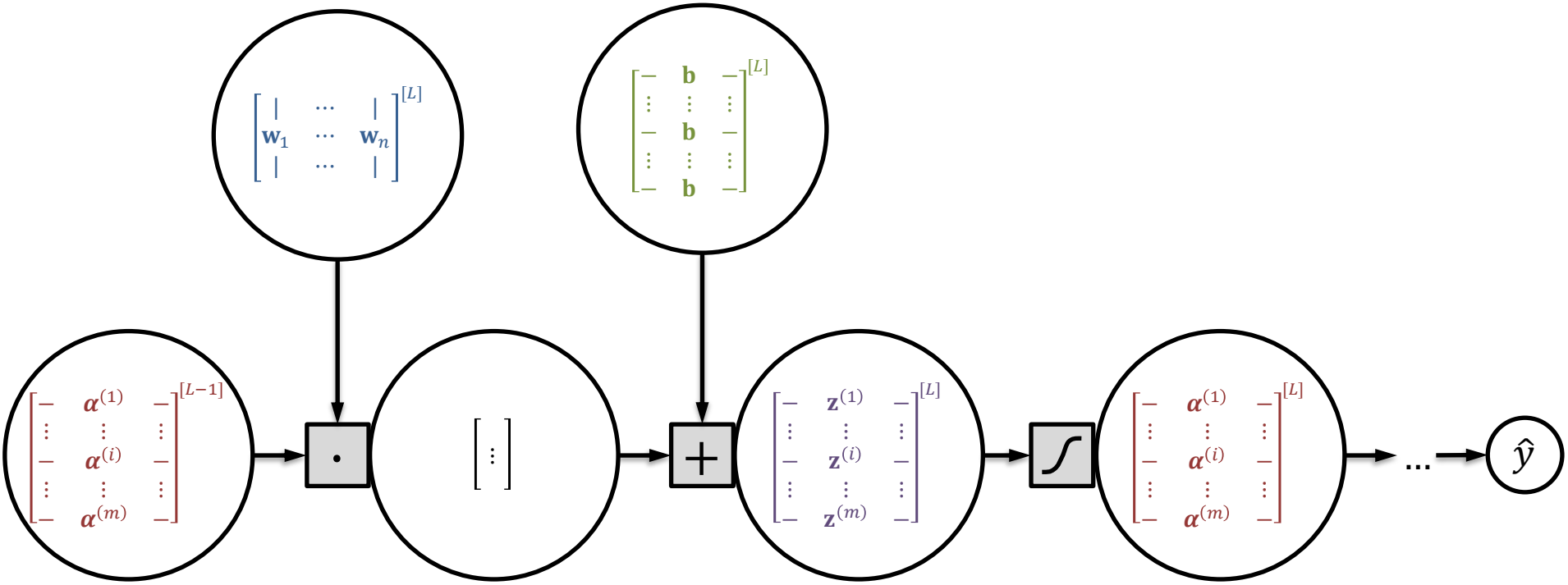
Introduction to Neural Networks (II)

Activations, Tasks, Losses and Architectures

Dimosthenis Karatzas (dimos@cvc.uab.es)

Getting gradient descent to work

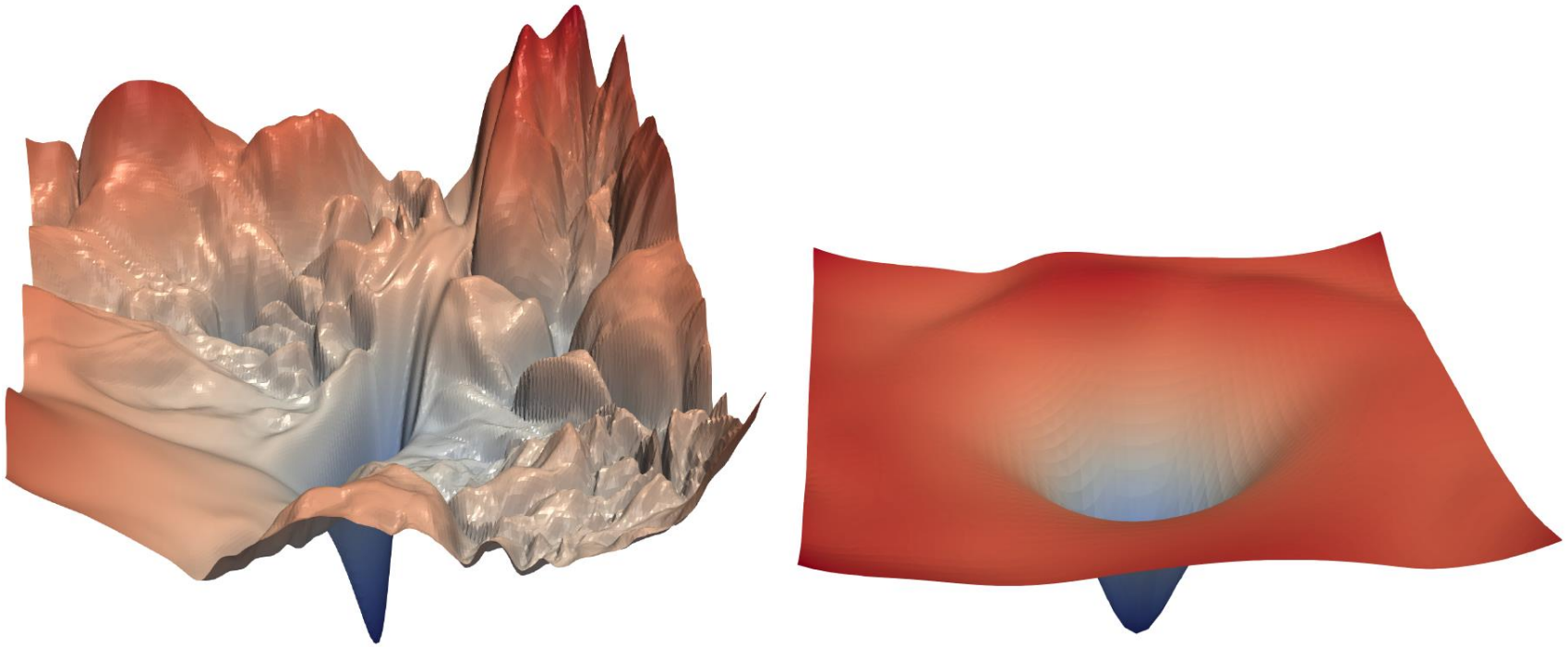
We know how to compute error derivatives for every weight given a complex model using **backpropagation** (and auto differentiation)



This is still far from having a mechanism that allows us to find good solutions

The Optimisation Landscape

Successfully training a neural network implies (1) **defining well-behaved loss landscapes** and (2) using the **right algorithms to tread through them**



Li et al "Visualizing the Loss Landscape of Neural Nets", NIPS 2018

Still Not A Learning algorithm

- We still need to understand
  - **Loss functions**: How to measure our error? This depends on the task we want to solve.
  - **Activation functions**: What kind of neurons should we use?
  - **Architectures**: How to combine neurons together to build meaningful models?
  - **Optimisation**: Is batch gradient descent the best way to use these error derivatives to discover a good set of weights?
  - **Regularisation**: How do we make sure we do not overfit?
  - **Initialisation**: Where do we start our search?

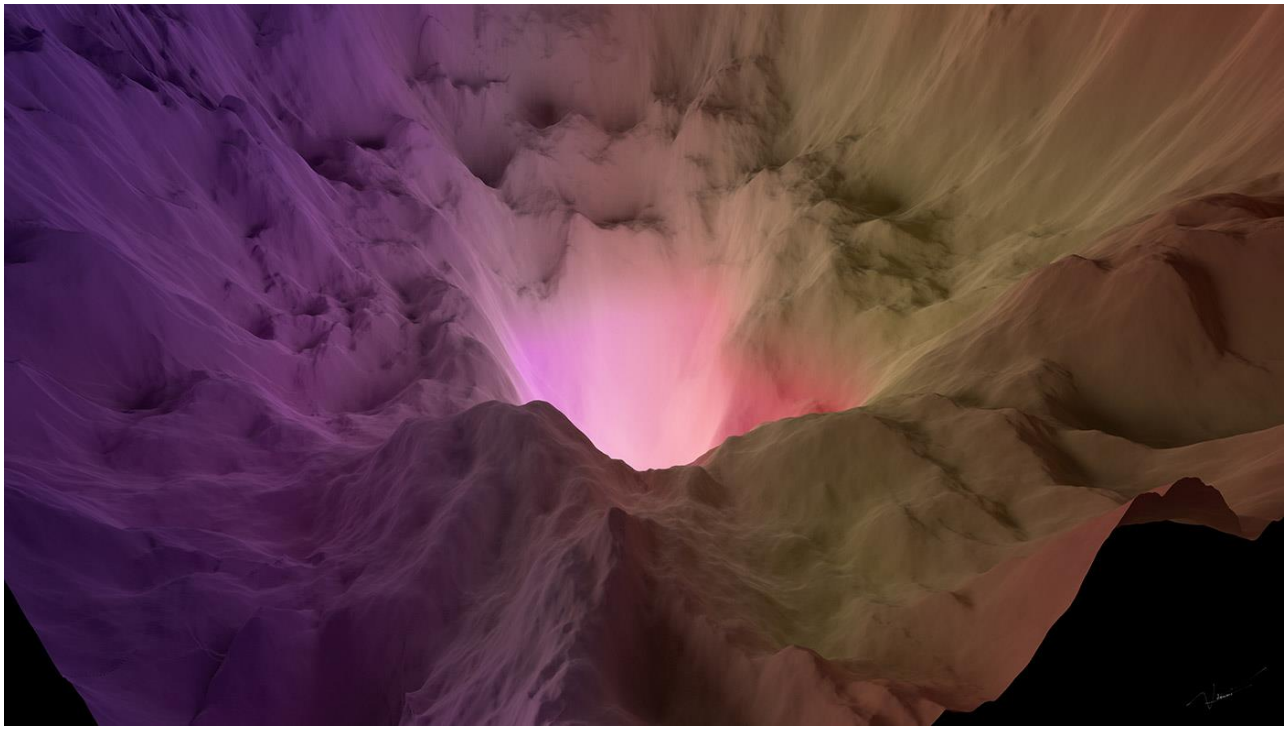
Define well-behaved landscapes

Efficient search

The Optimisation Landscape

Your landscape is defined by:

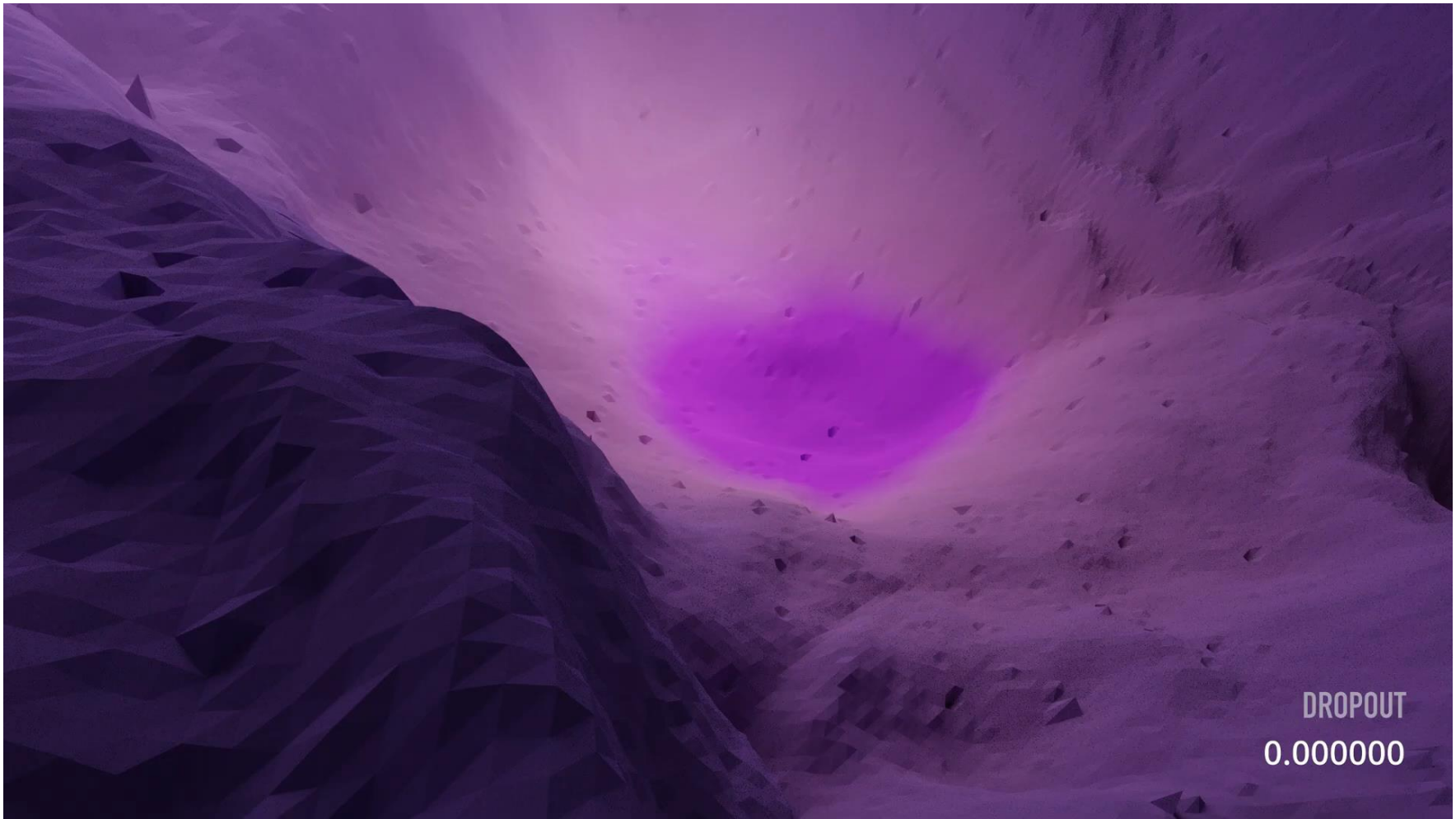
- Your model – the **architecture** and the **types of units** (activations) you use
- The **loss function** that you will employ
- Your **training set**



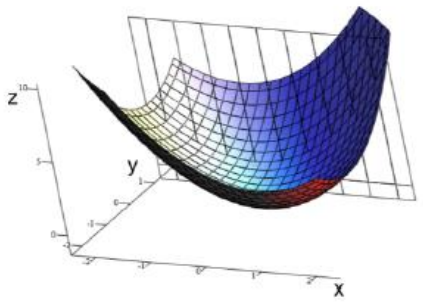
Loss Landscape visualization created with data from the training process of a convolutional network. Imaginette Dataset. Sgd-Adam, train mode, 1 million points, log scaled. ***Credit****: losslandscape.com*

The changing landscape

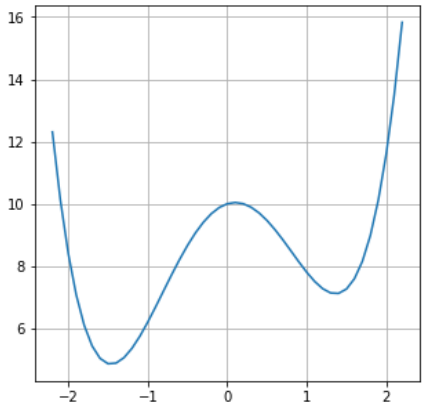
Moreover, your landscape is usually changing continuously, as you dynamically adjust aspects of your training process (more on this in the next lecture)



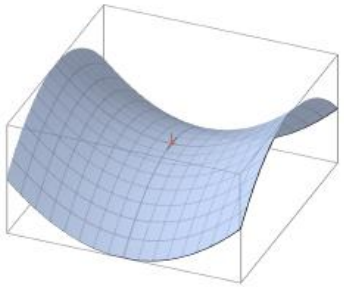
The Optimisation Landscape



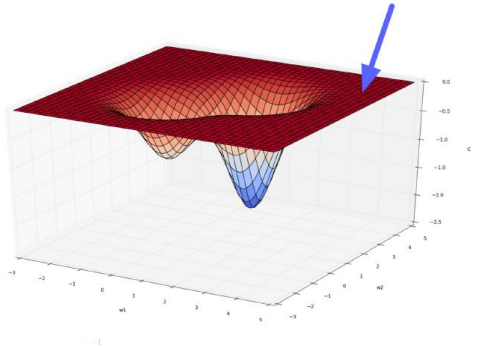
Convex functions



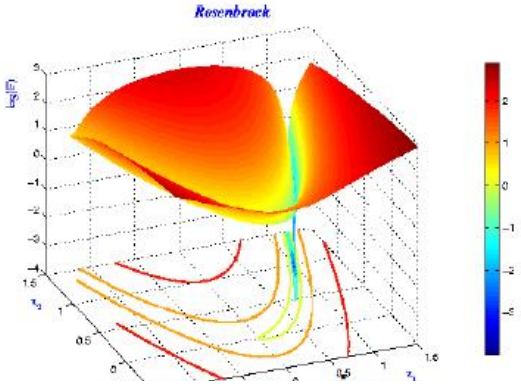
Local minima



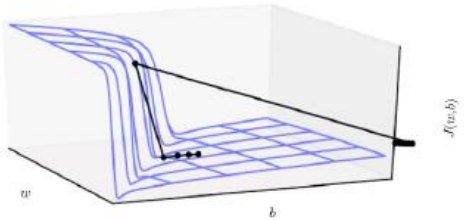
Saddle points



Plateaux



Ravines



Cliffs

Figure credit: J. Ba's course CSC413 on Neural Networks and Deep Learning

Which landscape?

STOCHASTIC VS BATCH LEARNING

Batch learning

So far, we have defined our cost function as the average loss over the whole set of training examples

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}, \mathbf{w}), y^{(i)})$$

Computing the gradient requires summing over **all** the training examples. Parameters are updated based on the gradient for the whole training set..

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \nabla_{\mathbf{w}} \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \frac{1}{m}\sum_{i=1}^{m} L^{(i)}(\mathbf{w})$$

This is called **batch training**.

Stochastic learning

Batch training is impractical with large datasets. Alternatively, one can use **stochastic (online) learning** where a single example is chosen from the training set at each iteration

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L^{(i)}(\mathbf{w})$$

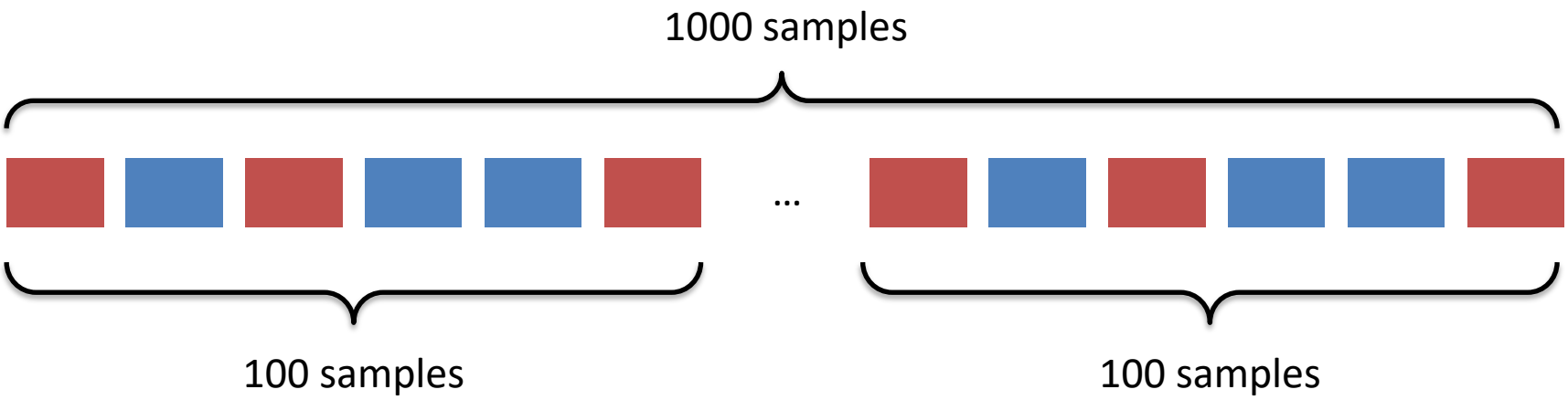
If training examples are sampled randomly, the stochastic gradient is an unbiased estimate of the batch gradient

$$\mathbb{E}_i\left[\nabla_{\mathbf{w}} J^{(i)}(\mathbf{w})\right] = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\mathbf{w}} J^{(i)}(\mathbf{w}) = \nabla_{\mathbf{w}} J(\mathbf{w})$$

A mental experiment

Stochastic learning can make significant progress before it has even looked at all the data

Imagine a dataset of 1000 samples composed of 10 identical copies of a set of 100 samples



The average gradient over the whole dataset is the same as the average gradient of the set of 100. In this scenario,

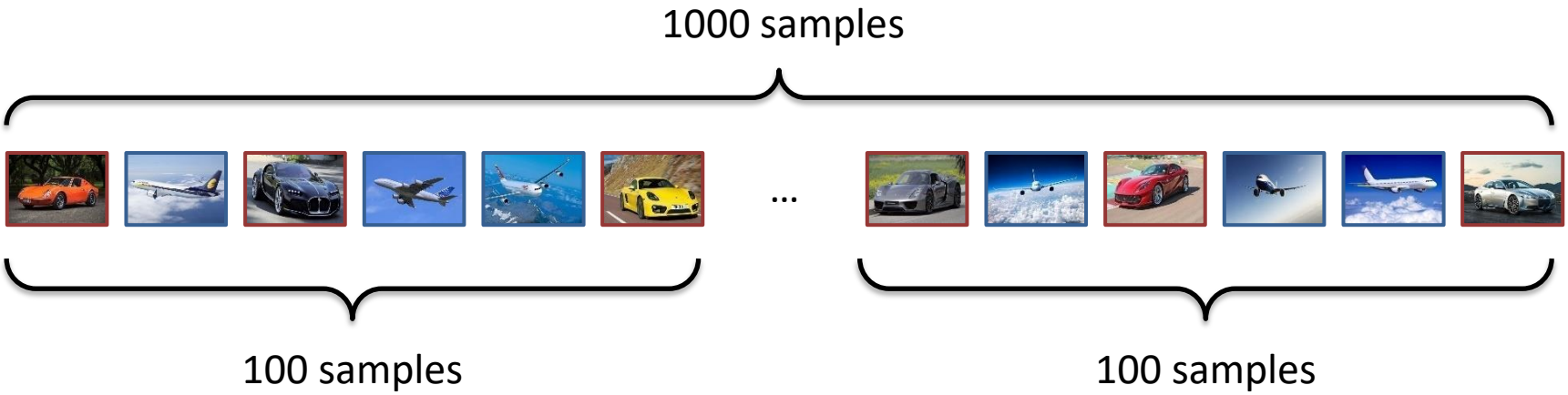
Batch gradient descent would calculate the same quantity 10 times over before one parameter update.

Stochastic gradient descent will view each epoch as 10 iterations through a 100-long training set.

A mental experiment

Stochastic learning can make significant progress before it has even looked at all the data

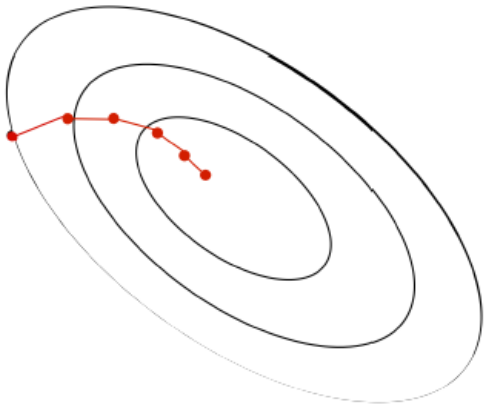
In practice, examples do not appear multiple times, but there are usually clusters of patterns that are very similar



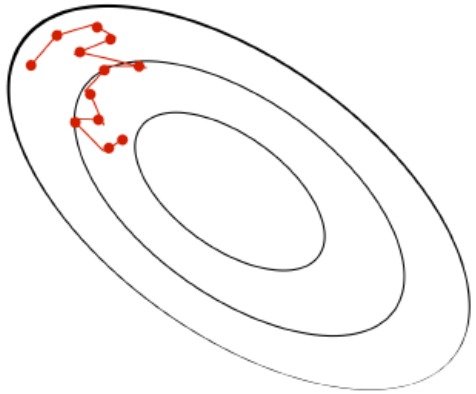
See also discussion in Y. LeCun et al "Efficient BackProp"

The Landscape of SGD

In Batch gradient descent you tread through the stable landscape defined by your whole training set. In Stochastic gradient descent, the landscape changes in every iteration!



Batch gradient descent



Stochastic gradient descent

The Landscape of SGD

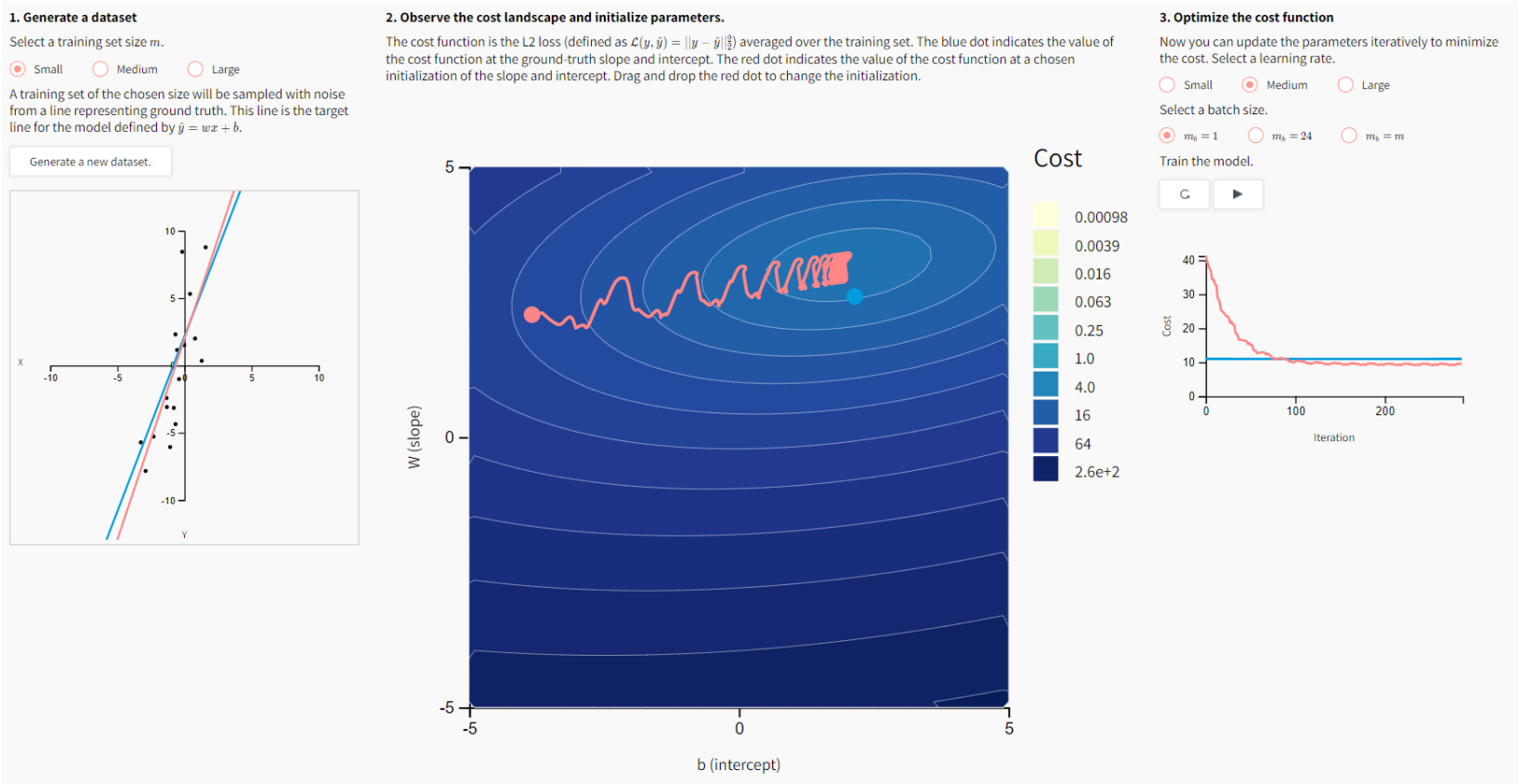
The noise actually helps! Batch gradient descent will fall into whatever basin it started from, but stochastic learning, due to the noise, can jump into the basin on another, possibly deeper, local minimum.

But, the noise also prevents SGD to fully converge to the (training set) minimum. SGD stalls out due to the weight fluctuations.

In order to reduce the fluctuations we can do two things:

- Decrease (anneal) the learning rate
- Introduce an adaptive batch size

Example

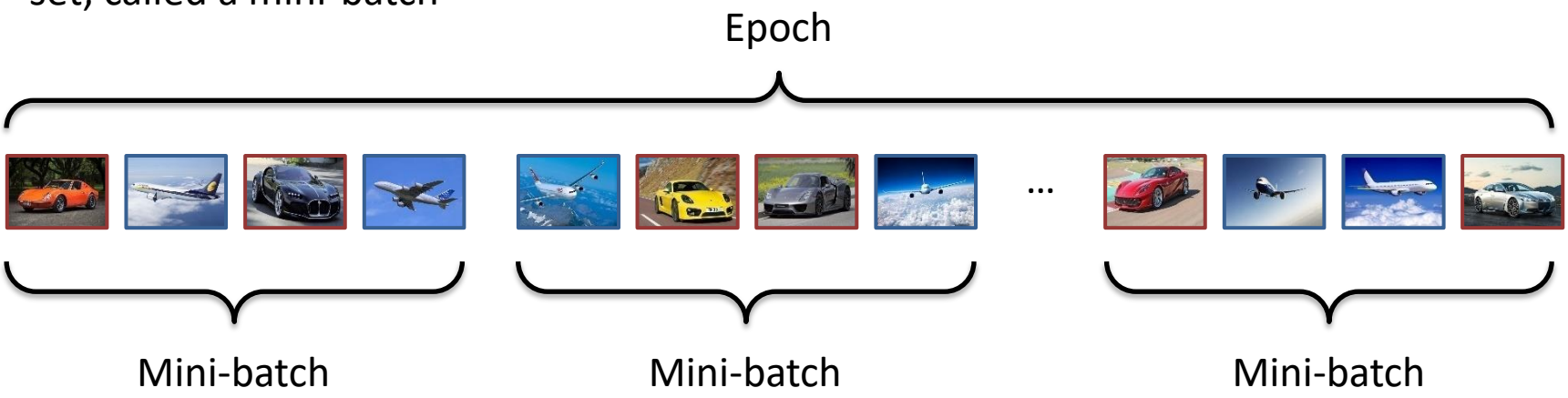


https://www.deeplearning.ai/ai-notes/optimization/

Mini-batches

Single sample SGD has to problems, it suffers from **fluctuations** and we cannot exploit efficient **parallel operations** of our CPU / GPU

A compromise is to compute the gradients on a medium-sized subset of our training set, called a mini-batch

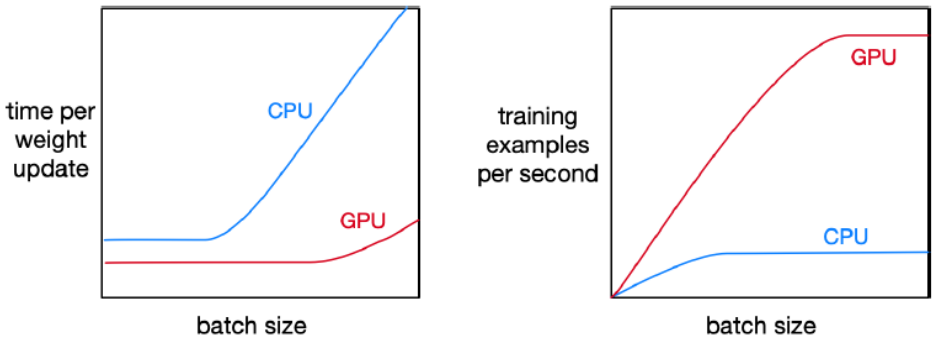


Larger mini-batches result to smaller variance, inversely proportional to the mini-batch size

The right mini-batch size

Large batches converge in less steps because each update is less noisy, but take longer to calculate
Small batches need more steps to converge, but result to more weight updates per second as they require less computation

But, with parallelisation techniques (vectorised operations), an update with batch size S = 10 is not much more expensive than an update with batch size S = 1

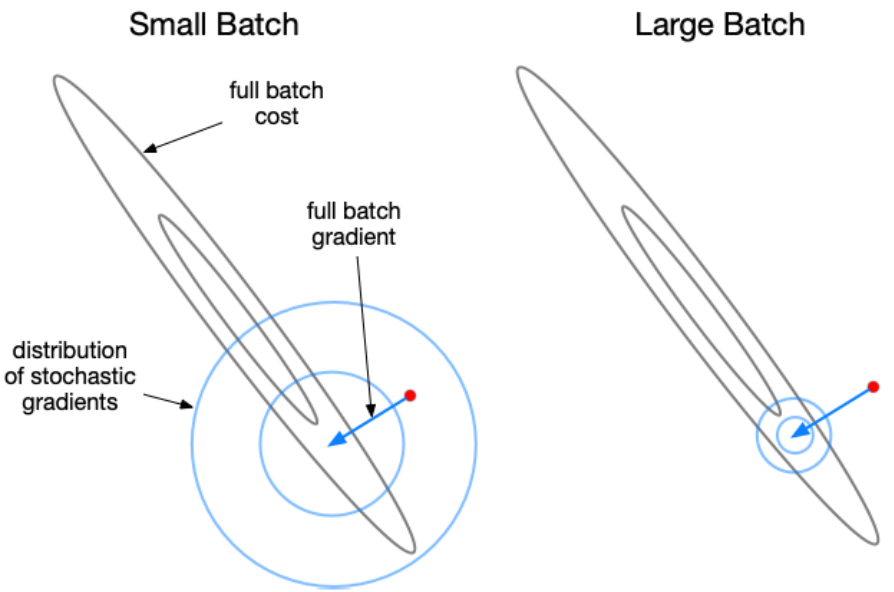


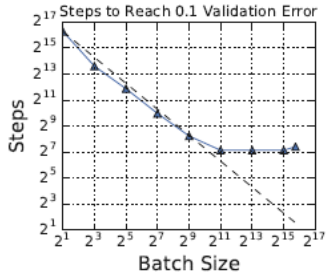
The right choice depends, among others, on available equipment. Rule of thumb: choose the largest mini-batch size that does not saturate your hardware

Figure credit: J. Ba's course CSC413 on Neural Networks and Deep Learning

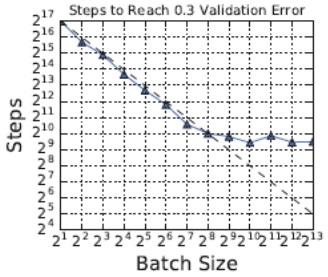
The right mini-batch

After some size, mini-batches approximate the lower variance of the full batch. Further increase in size, does not help much.

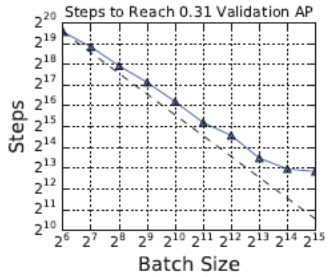




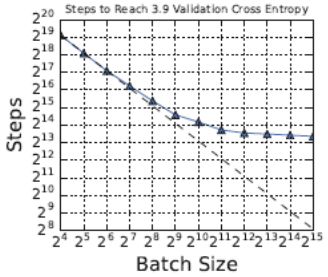
(b) Simple CNN on Fashion MNIST



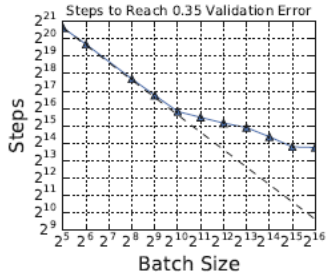
(c) ResNet-8 on CIFAR-10



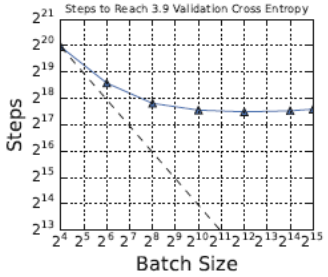
(e) ResNet-50 on Open Images



(f) Transformer on LM1B



(h) VGG-11 on ImageNet



(i) LSTM on LM1B

Figure credit: Shallue et.al. "Measuring the effects of Data Parallelism on Neural Network Training" JMLR2019

Shuffling and balancing

In Batch learning, the order of input presentation is irrelevant – it sees all the information at the same time.

In Stochastic learning, the order is important

- We could in principle emphasize examples (e.g. to boost the performance for infrequently occurring inputs)
- See for example focal loss (weights the contribution of each sample to the loss based in the classification error) or the use of hard negatives in metric learning losses

Good practice:

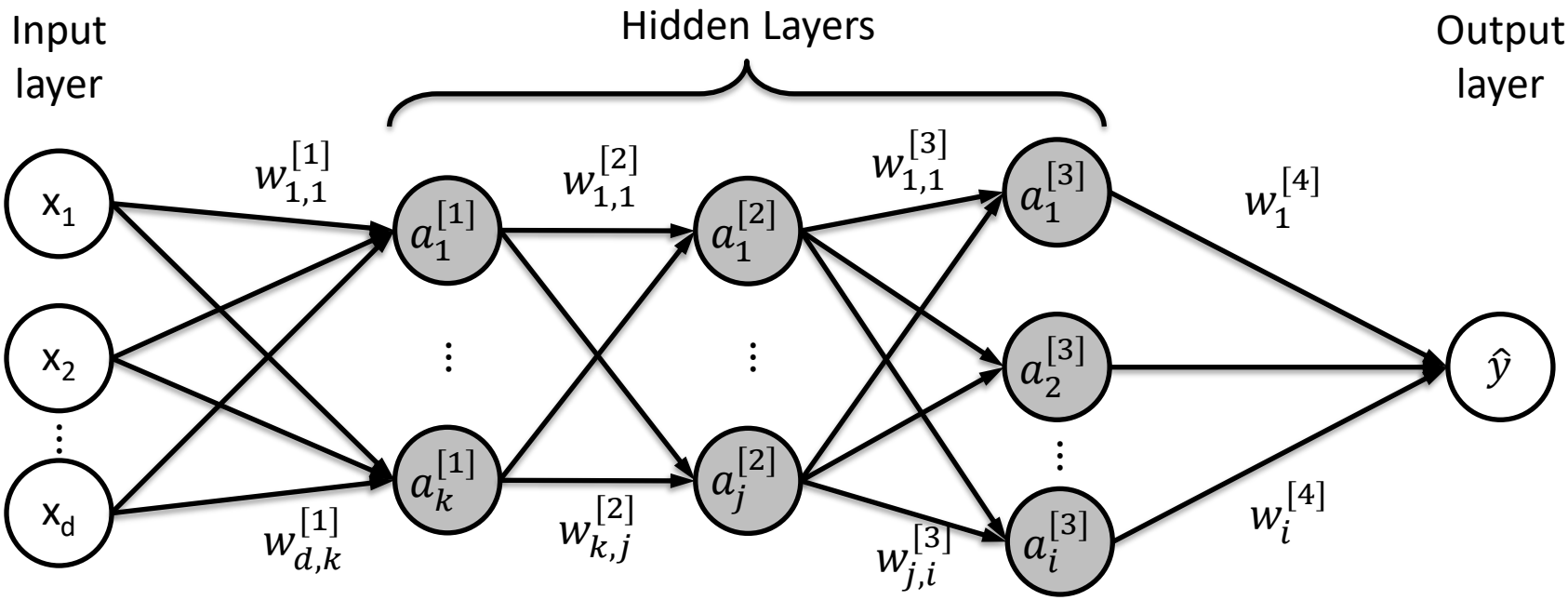
- **Balance** datasets beforehand (when possible)
- **Shuffle** in every epoch
    - making sure all mini-batches are uniformly sampled from the distribution and
    - all the samples are seen once in every epoch

Stochastic vs Batch

- Stochastic
  - Usually much faster
  - Noise often results in better solutions, but prevents full convergence to the empirical minimum (of the training set)
  - Can be used for tracking changes (problems changing over time)
- Batch
  - Convergence conditions are well understood
  - Certain acceleration techniques only operate in batch learning
  - Theoretical analysis is simpler
  - No noise – ensures convergence to the empirical minimum (of the training set)

ACTIVATION FUNCTIONS

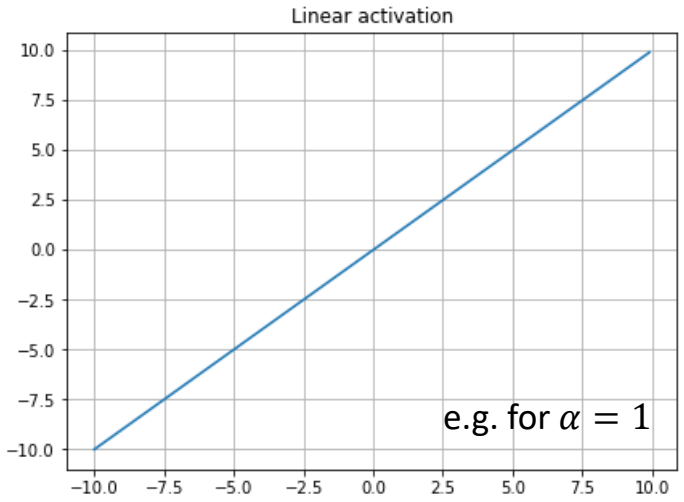
What activations to use?

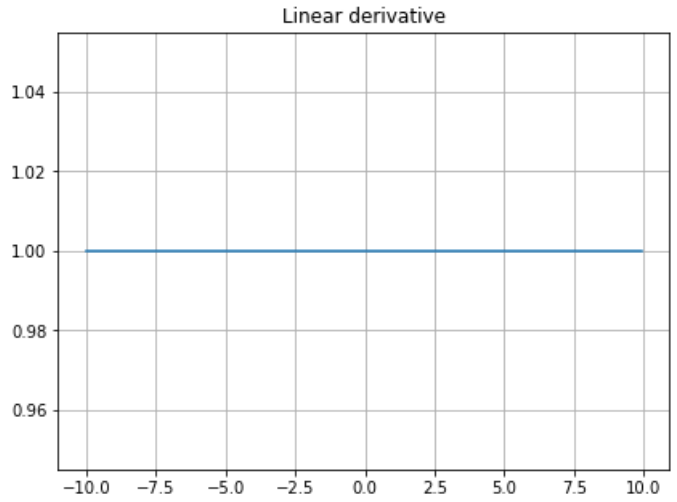


Linear Neurons

$$\hat{y} = \alpha * z$$

$$\frac{d\hat{y}}{dz} = \alpha$$





Any sequence of linear layers can be equivalently represented by a single linear layer

$$\hat{\mathbf{y}} = \underbrace{\mathbf{W}^{[3]}\mathbf{W}^{[2]}\,\mathbf{W}^{[1]}}\mathbf{x}$$

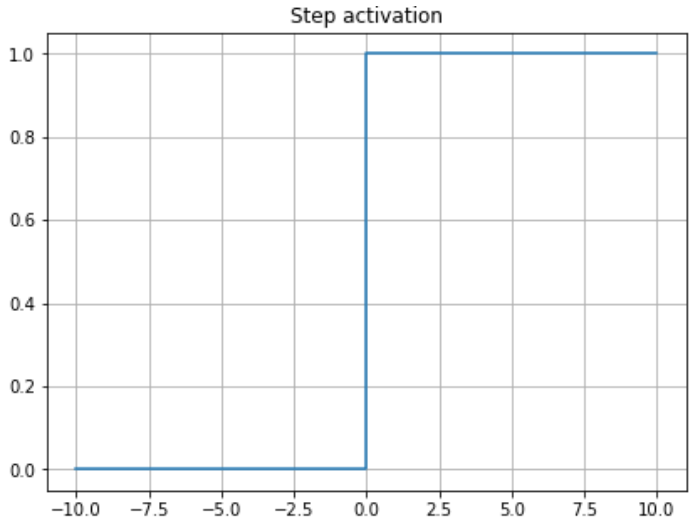
$$\triangleq \mathbf{W}'$$

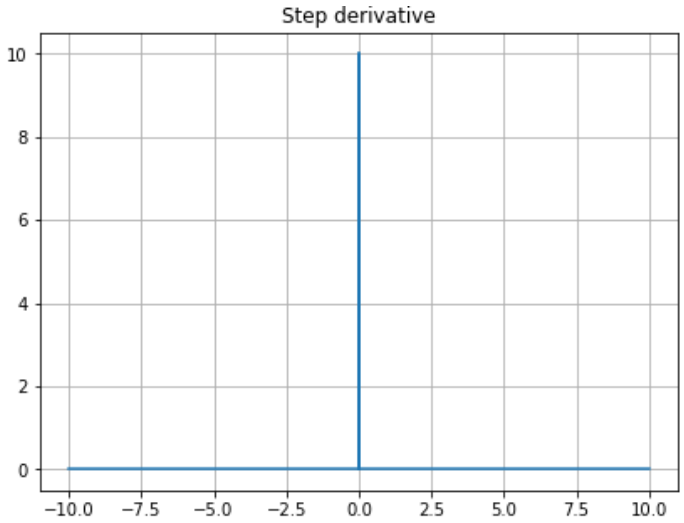
A linear activation function is a bad idea! - Deep linear networks are no more expressive than linear regression

Binary Threshold Neurons

$$T(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$T'(z) = \begin{cases} 0 & z \neq 0 \\ \infty & z = 0 \end{cases}$$





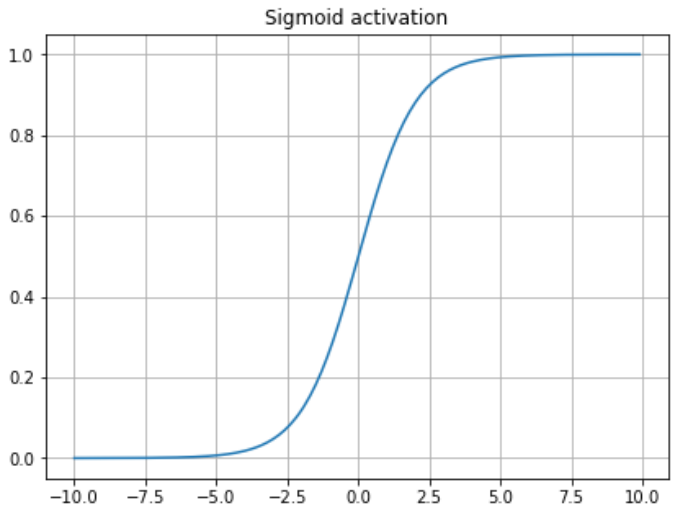
Binary threshold units do not produce useful derivatives

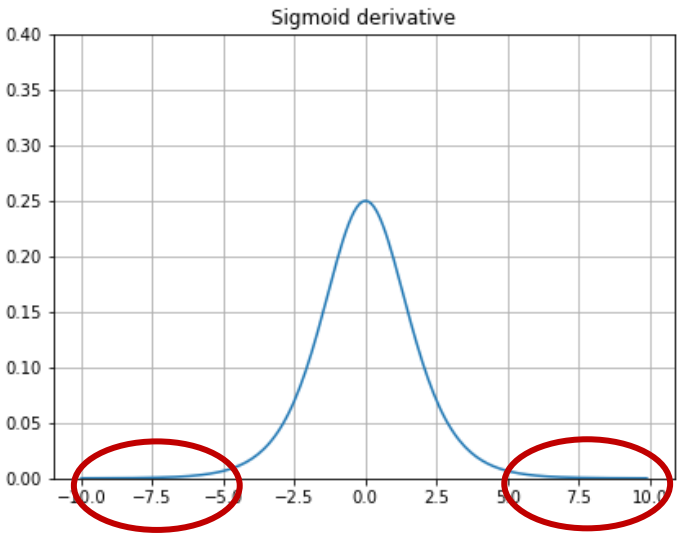
$$\frac{d}{dw_i} L(\hat{y}, y) = \frac{dL}{d\hat{y}} \frac{dT(z)}{dz} \frac{dz}{dw_i}$$

Logistic (Sigmoid) Activation

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$





$$\frac{d}{dw_i} L(\hat{y}, y) = \frac{dL}{d\hat{y}} \frac{d\sigma(z)}{dz} \frac{dz}{dw_i}$$

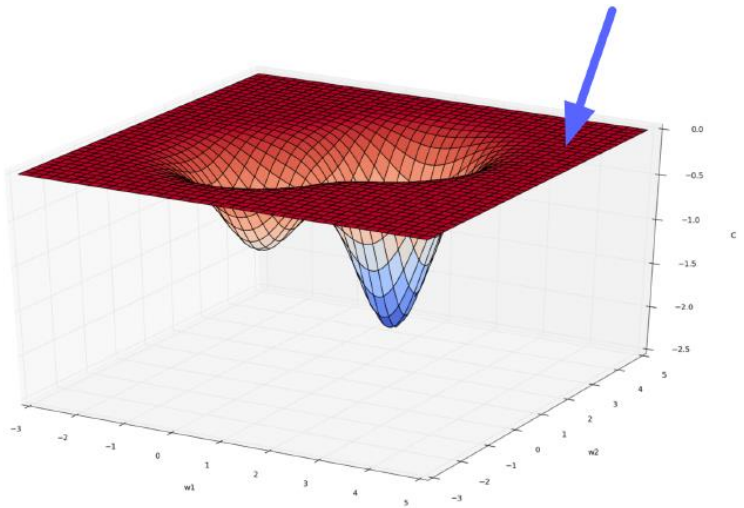
One problem with the logistic function is that when it is near the edge of its dynamic range, the derivative is very small.

Saturated Units

the activation functions that are finite at both ends of their outputs (like the family of sigmoid functions) are called **saturated activation functions**. Functions which are infinite on at least one of their ends, are **non-saturated activation functions**.

Units whose activations are always near the ends of their dynamic range are called **saturated units**.

Saturated units kill the gradient. They give rise to plateaux in the loss landscape which are difficult to escape from



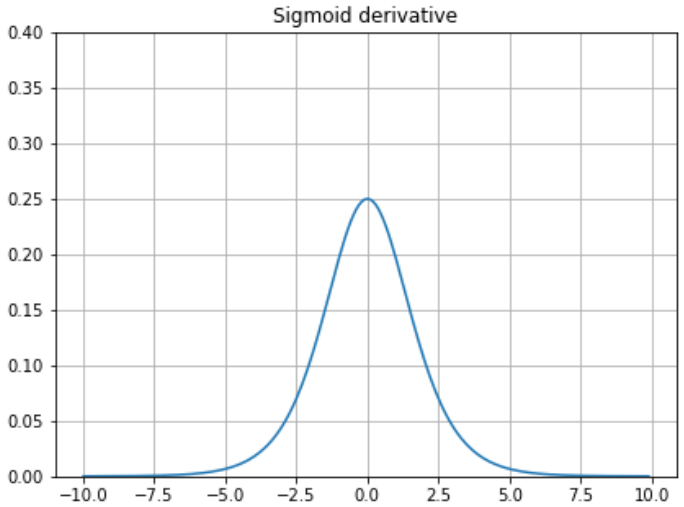
Vanishing gradients

Even if the logistic unit functions near the zero area, the maximum value for the derivative is 0.25 for z=0.

The gradient tends to get smaller as we move backwards through the hidden layers

Neurons in the earlier layers learn much more slowly than neurons in later layers

$$\frac{d\hat{y}}{dz} = \sigma(z)(1-\sigma(z))$$



$$\sigma'(z_1)\, w_2\, \sigma'(z_2)\, w_3\, \sigma'(z_3)\, w_4\, \sigma'(z_4) \frac{\partial L}{\partial \alpha_4} = \frac{\partial L}{\partial b_1}$$

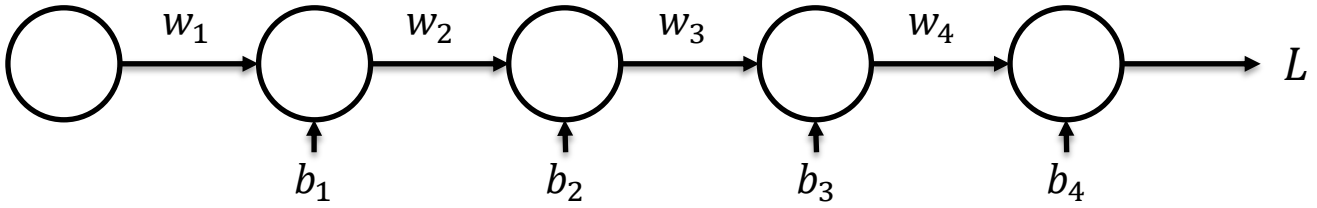
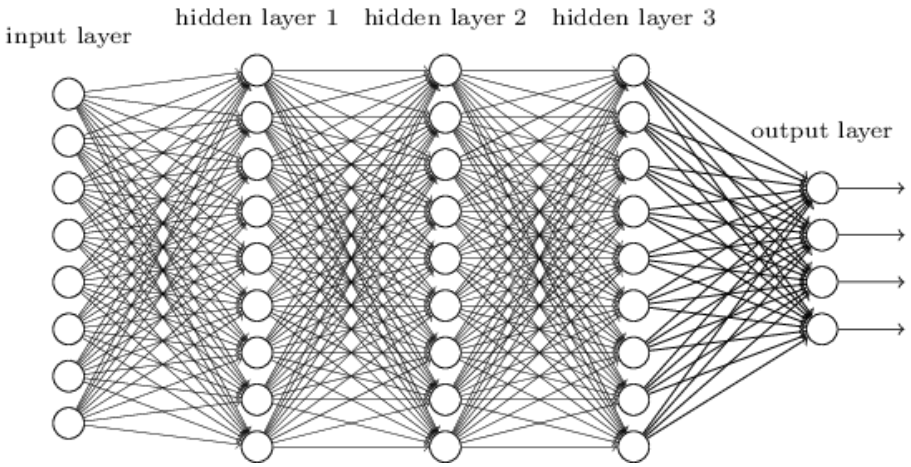
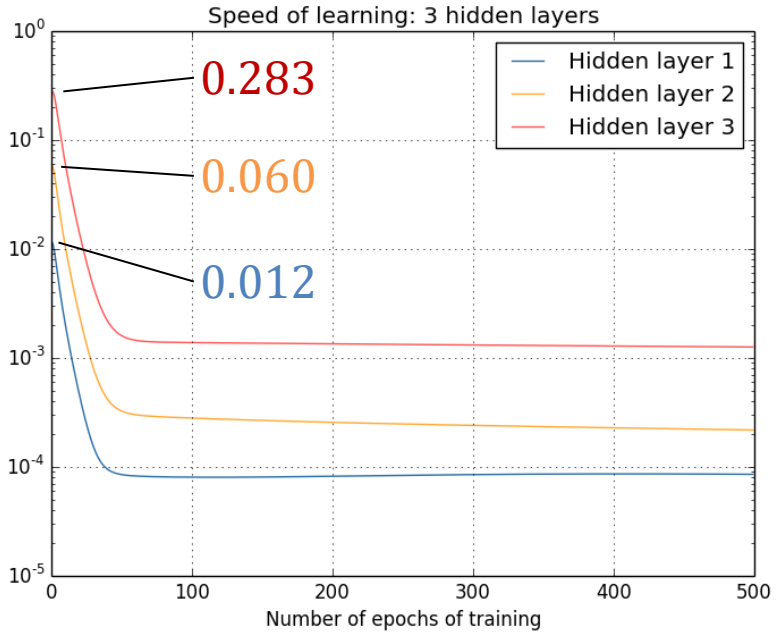


Figure by Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap5.html)

Vanishing gradients



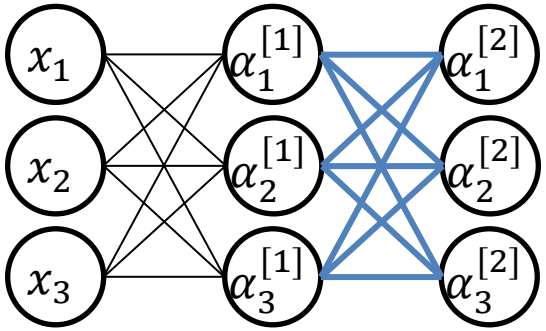


Vanishing gradients cause units in the earlier layers learn much more slowly than units in later layers.

Analysis by Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap5.html)

Non-Zero Centered Activations

Logistic outputs are not zero-centred – they are always positive



Logistic units

All positive activations

All positive input to the next layer

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]T}\boldsymbol{\alpha}^{[1]} + \mathbf{b}^{[2]}$$

$$\boldsymbol{a}^{[2]} = \sigma\left(\mathbf{z}^{[2]}\right)$$

$$\frac{d}{d\mathbf{W}^{[2]}} L(\hat{y}, y) = \frac{dL}{d\mathbf{z}^{[2]}} \frac{d\mathbf{z}^{[2]}}{d\mathbf{W}^{[2]}}$$

$\boldsymbol{\alpha}^{[1]}$

All elements positive

Either all elements positive or all negative

Non-Zero Centered Activations

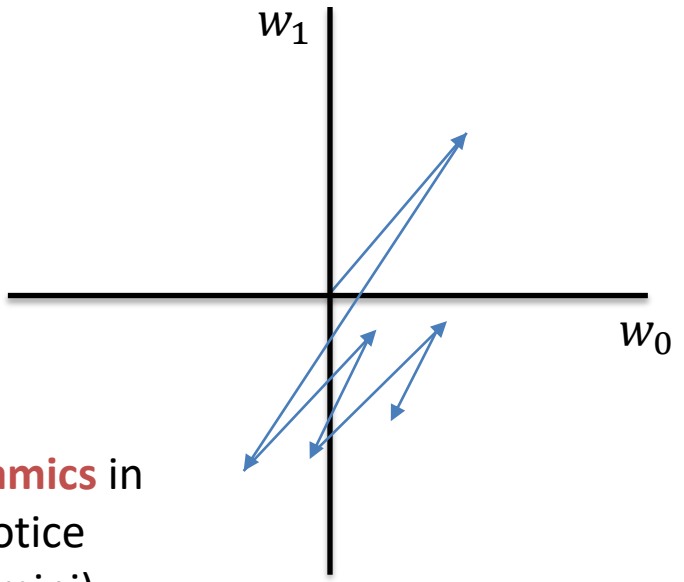
$$\frac{d}{d\mathbf{W}^{[2]}} L(\hat{y}, y) = \frac{dL}{d\mathbf{z}^{[2]}} \frac{d\mathbf{z}^{[2]}}{d\mathbf{W}^{[2]}}$$

$\boldsymbol{\alpha}^{[1]}$

All elements positive

Either all elements positive or all negative

$$\mathbf{W}^{[2]} \leftarrow \mathbf{W}^{[2]} - \underbrace{\alpha \frac{d}{d\mathbf{W}^{[2]}} L}_{\delta \mathbf{W}}$$

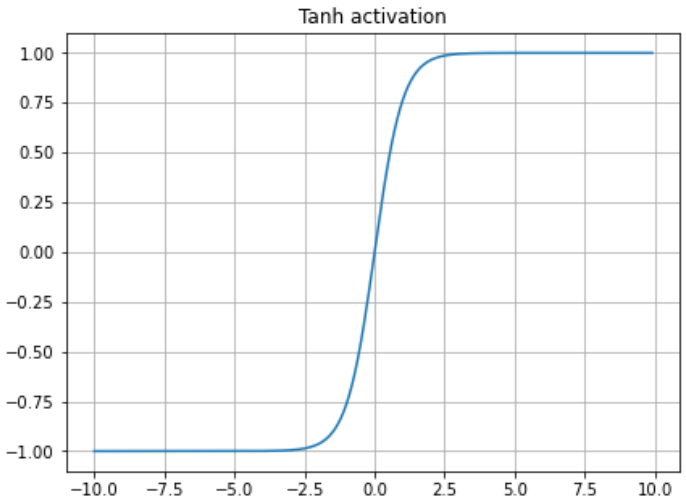


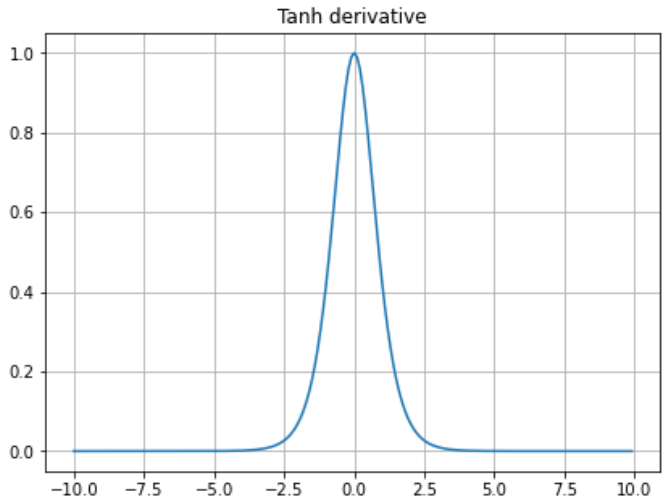
This could introduce undesirable **zig-zagging dynamics** in the gradient updates for the weights. However, notice that once these gradients are added up across a (mini) batch of data the final update for the weights can have variable signs, somewhat mitigating this issue.

Tanh Activation

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh'(z) = 1 - \tanh^2(z)$$





Tanh is just a rescaled and shifted version of the Sigmoid function.

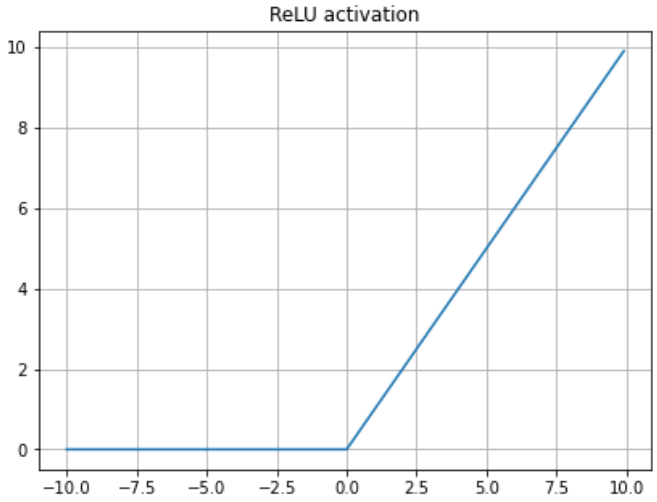
Output is now zero centred, in [-1, 1]

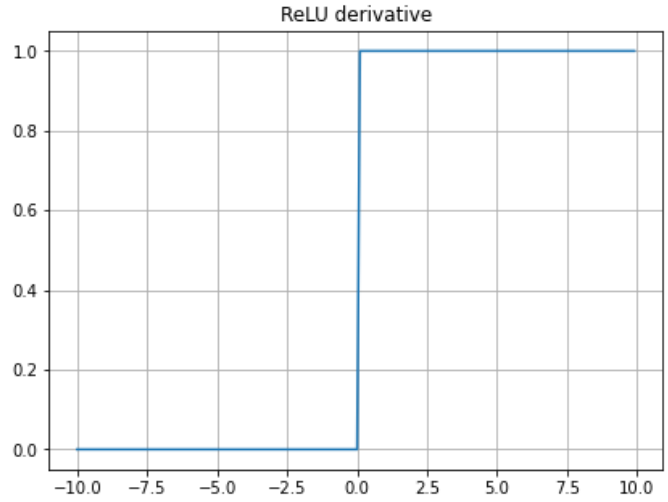
But the Tanh also saturates like the logistic

$$\begin{aligned}\tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{(e^z - e^{-z})(e^z)}{e^z + e^{-z}} \\ &= \frac{e^{2z} - 1}{e^{2z} + 1} = 2\frac{e^{2z}}{e^{2z} + 1} - 1 \\ &= 2\frac{1}{\frac{e^{2z} + 1}{e^{2z}}} - 1 = 2\frac{1}{1 - e^{-2z}} \\ &= 2\sigma(2z) - 1\end{aligned}$$

ReLU Activation

$$\mathrm{ReLU}(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases} \qquad \mathrm{ReLU}'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$





ReLU does not saturate for positive z, and it is sparsely activated

But, ReLU units can die if z is consistently negative. **Dead units** are units whose activations are always very close to zero

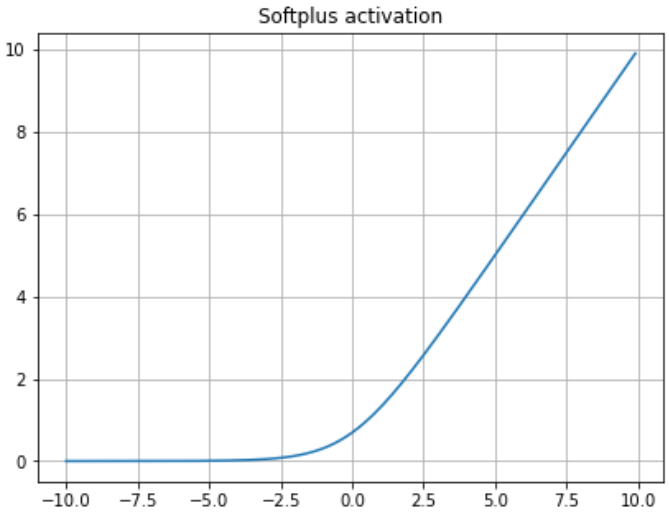
One workaround is to initialize these units with a small positive bias (i.e. 0.1). But still, we have to control the dynamics of the learning.

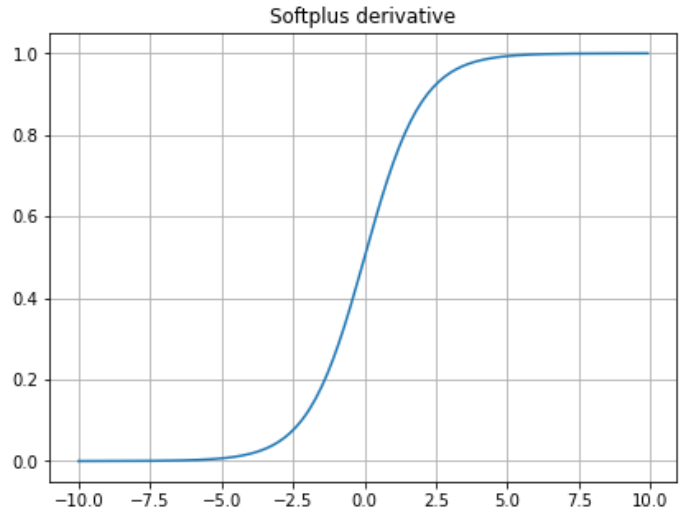
V. Nair and G. E. Hinton. "Rectified linear units improve restricted Boltzmann machines". ICML 2010

Softplus

$$softplus(z) = \ln(1 + e^z)$$

$$softplus'(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$





Softplus is a derivable alternative of the ReLU (actually, much older)

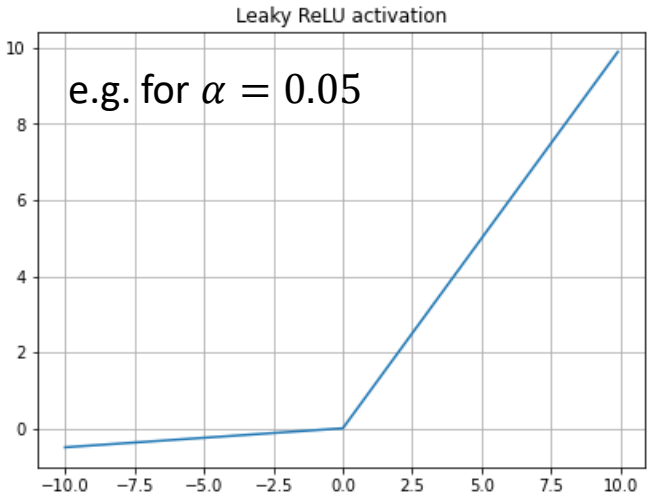
The derivative of the softplus is the logistic function

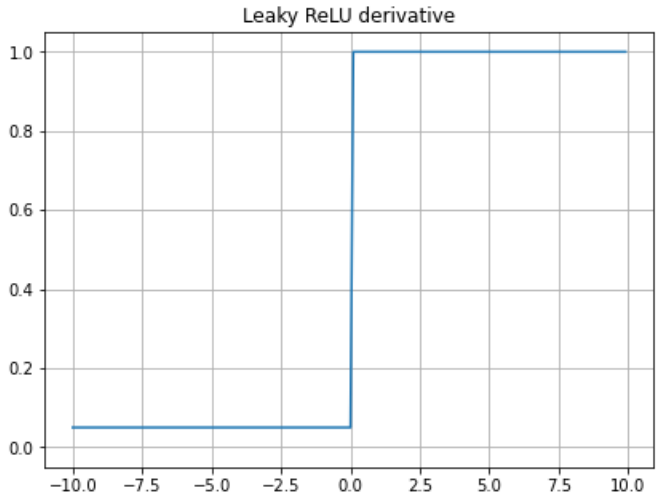
C. Dugas, "Incorporating Second-Order Functional Knowledge for Better Option Pricing", 2001

Leaky and Parametric ReLU

$$\mathrm{R}(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$$

$$\mathrm{R}'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$$





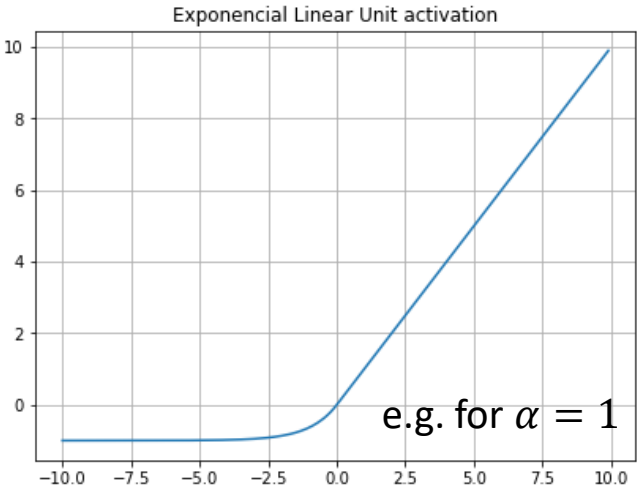
Leaky ReLU prevents the dying ReLU problem

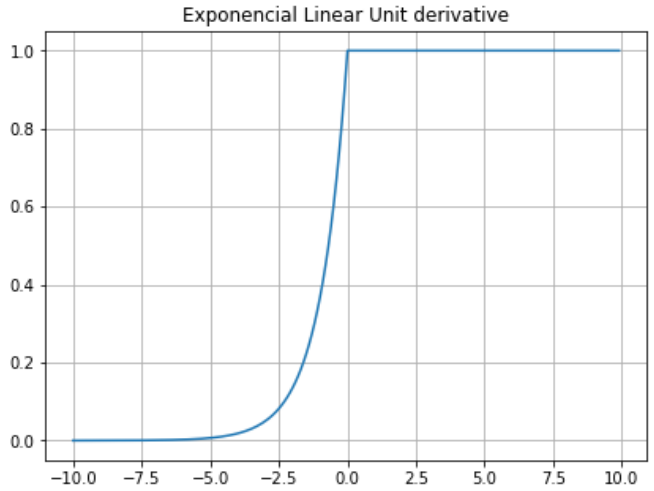
In the case of Parametric ReLU the slope α of the negative branch becomes a parameter of the model that can be learnt

Unlike sigmoid functions and its ReLU counterpart, Leaky ReLU does not provide consistent predictions for negative input values

Exponential Linear (ELU, SELU)

$$\mathrm{ELU}(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases} \qquad \mathrm{ELU}'(z) = \begin{cases} 1 & z \geq 0 \\ \alpha e^z & z < 0 \end{cases}$$





Similar to leaky ReLU, ELU has a small slope for negative values. Instead of a straight line, it uses a log curve

It saturates for large negative values, allowing them to be essentially inactive

Clevert et al, "Fast and Accurate Deep Network Learning by Exponential Linear Units", ICLR 2016

ReLU vs Sigmoid

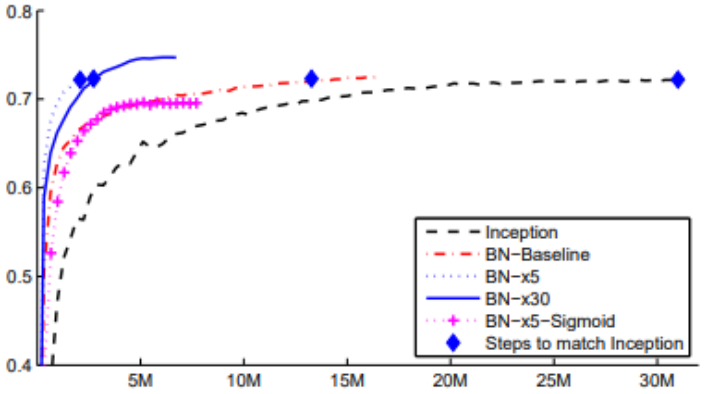


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

The performance of deep networks with Sigmoid nonlinearity can reach the ones using ReLU if proper normalization is used.

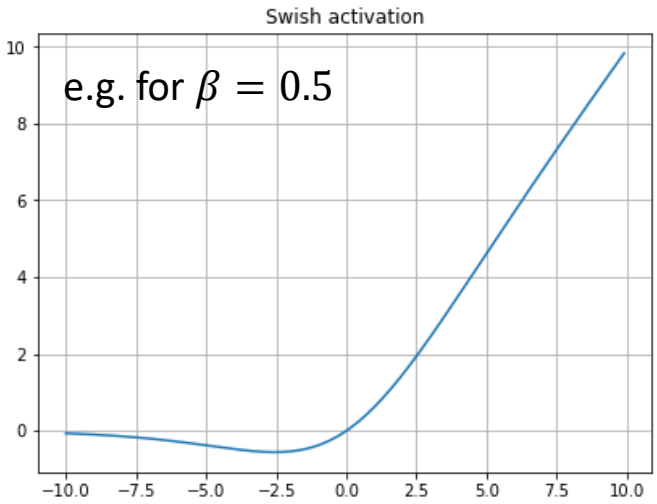
Results above are for the Inception model on LSVRC 2012. The accuracy for Sigmoid (logistic) is just slightly lower than ReLU's (69.8% vs 73.0%). Without normalization the accuracy was always below 0.1%

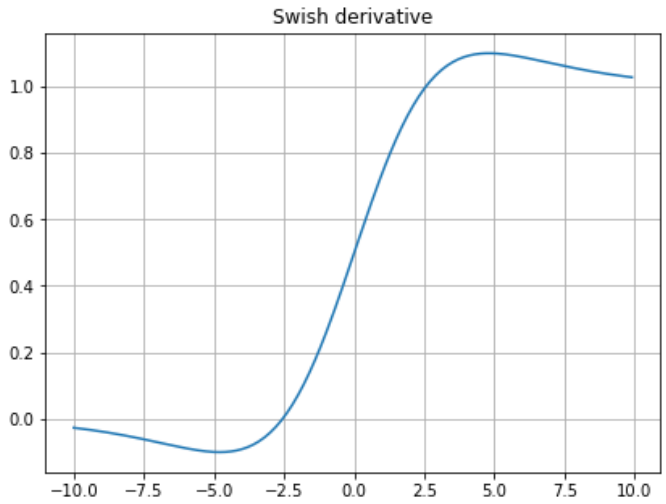
Don't take things for granted – try things out

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Swish

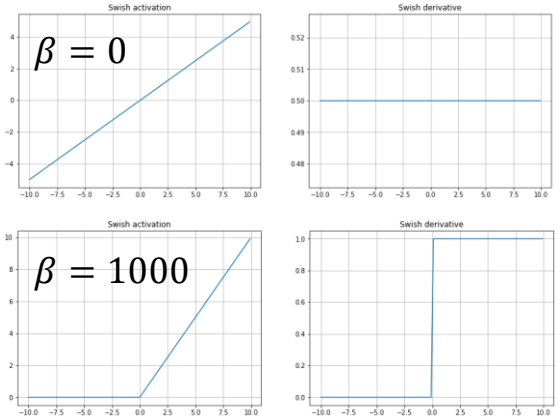
$$S(z) = z\,\sigma(\beta z) = \frac{z}{1 + e^{-\beta z}} \qquad S'(z) = \beta\,S(z) + \sigma(\beta z)(1 - \beta\,S(z))$$





Swish is a simple modification of the sigmoid. The parameter β is either constant or trainable

When $\beta = 0$, Swish becomes a scaled linear function. When β tends to ∞, Swish becomes a ReLU



P. Ramachadran et al, "Searching for activation functions", 2017

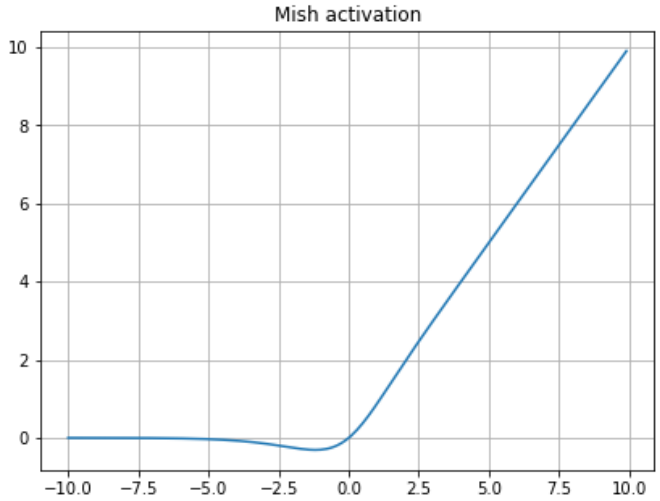
Mish

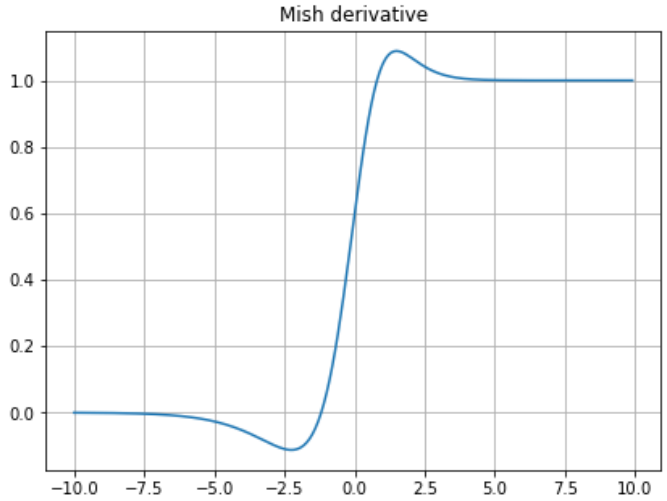
$$M(z) = z \tanh(softplus(z))$$

$$M'(z) = \frac{e^z \omega}{\delta^2}$$

$$\omega = e^{3z} + 4e^{2z} + (6 + 4z)e^z + 4(z + 1)$$

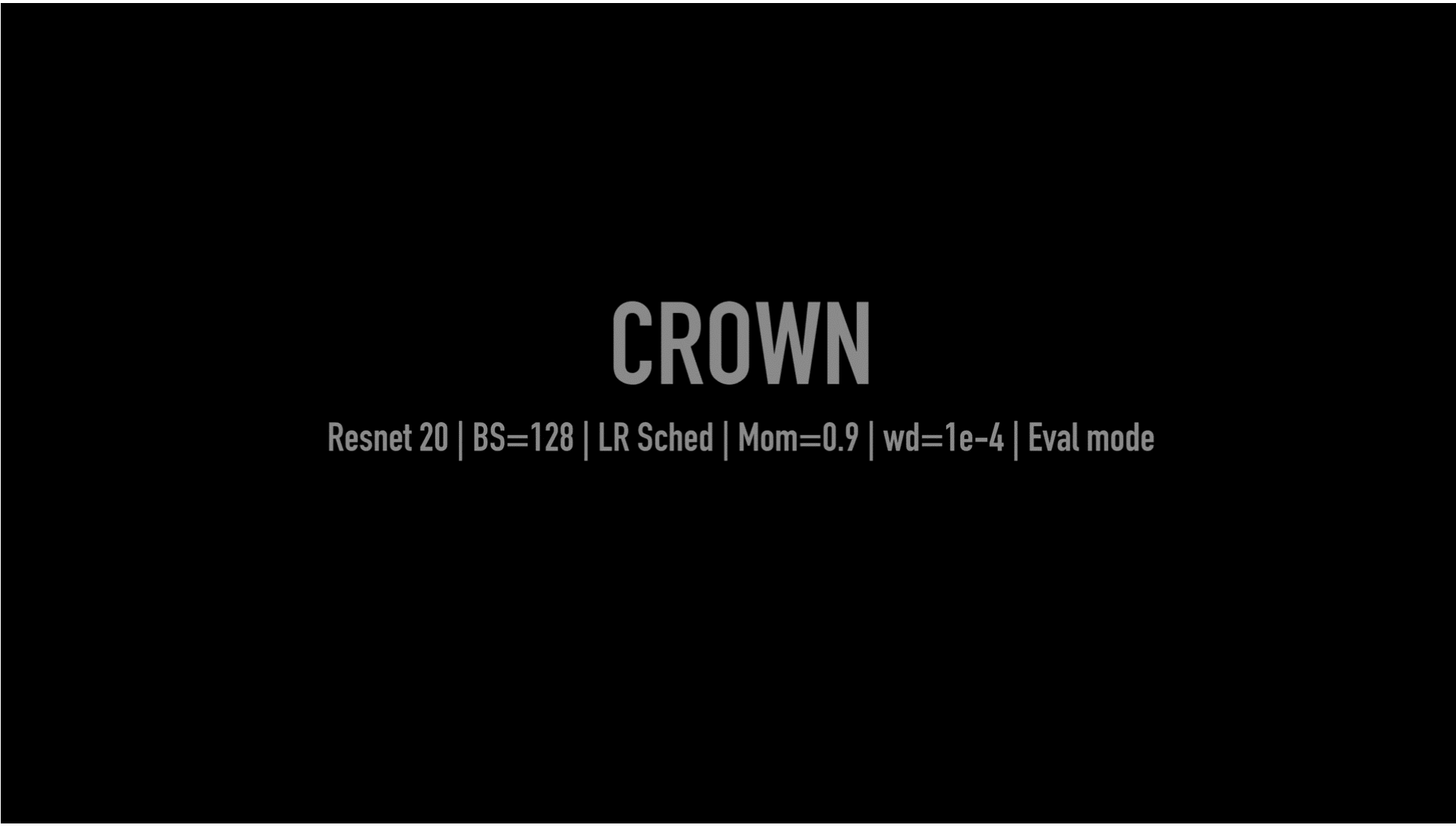
$$\delta = (e^z + 1)^2 + 1$$





Mish: A Self Regularized Non-Monotonic Activation Function, BMVC 2020

ReLU comparison with newcomers



General Advice

There is no easy way to choose your activation functions, you should **try out different alternatives** until you find what works best for the problem at hand

This analysis is not supposed to offer you any easy rule of thumb, but to highlight different options and give you some background that should help you debug/babysit your net

Be careful, among others, with:

- **Dead and saturated units**
  - They kill your gradient flow either because their gradients (saturated units) or their activation values (dead units) are close/equal to zero
- **Vanishing (and exploding) gradients**
  - Can cause (very) slow learning for the early units
- **Non-zero centred activations**
  - Can cause zig-zagging, and slower convergence

Most of the above problems can be alleviated with proper **initialization**, proper **normalization** and a proper **mini-batch size**, as well as choosing a good **optimization** method (see next week's lecture)

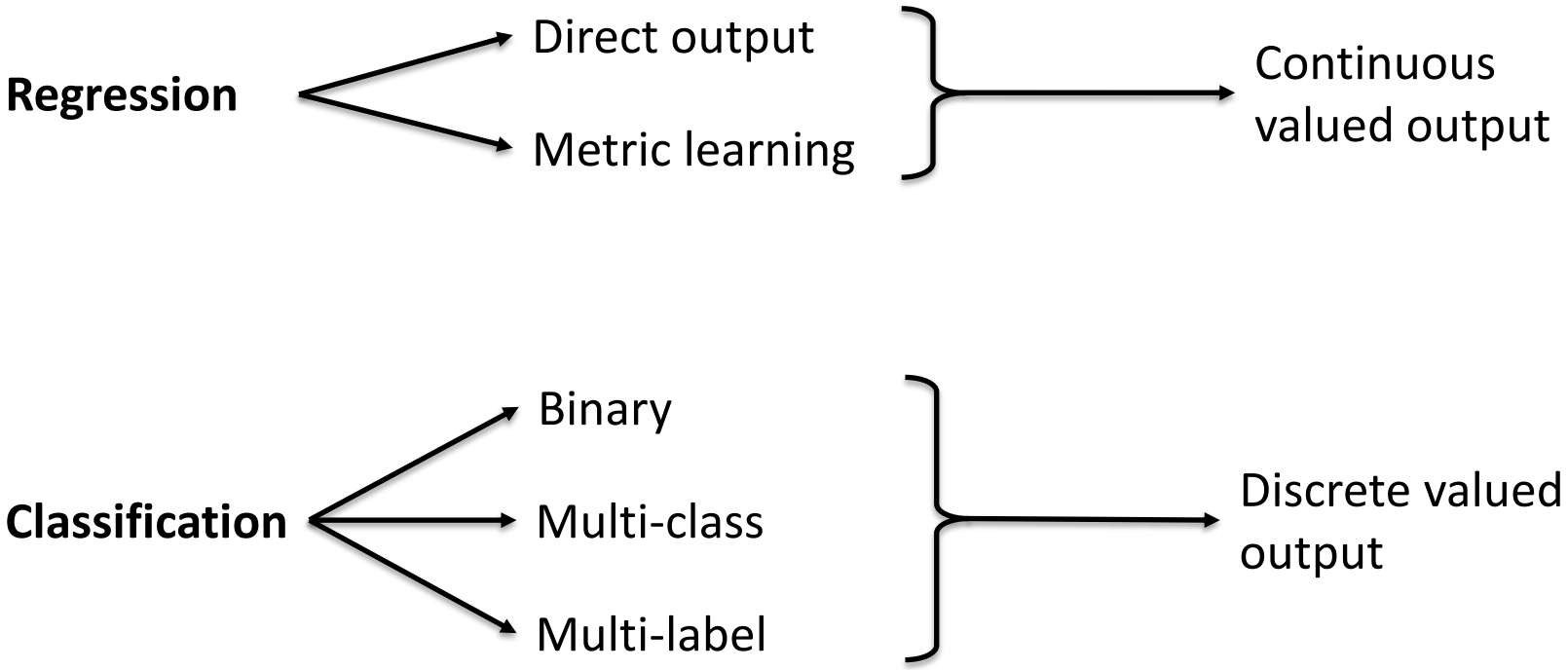
Note that we need (at least half) saturated functions for NNs to be universal approximators

TASKS, LOSSES AND THE OUTPUT LAYER

Types of learning tasks

- Supervised learning
  - Learn to predict an output when given an input vector.
  - **Regression**
  - **Classification**
- Unsupervised learning
  - Discover a good **internal representation** of the input
- Reinforcement learning
  - Learn to select an action to maximize payoff

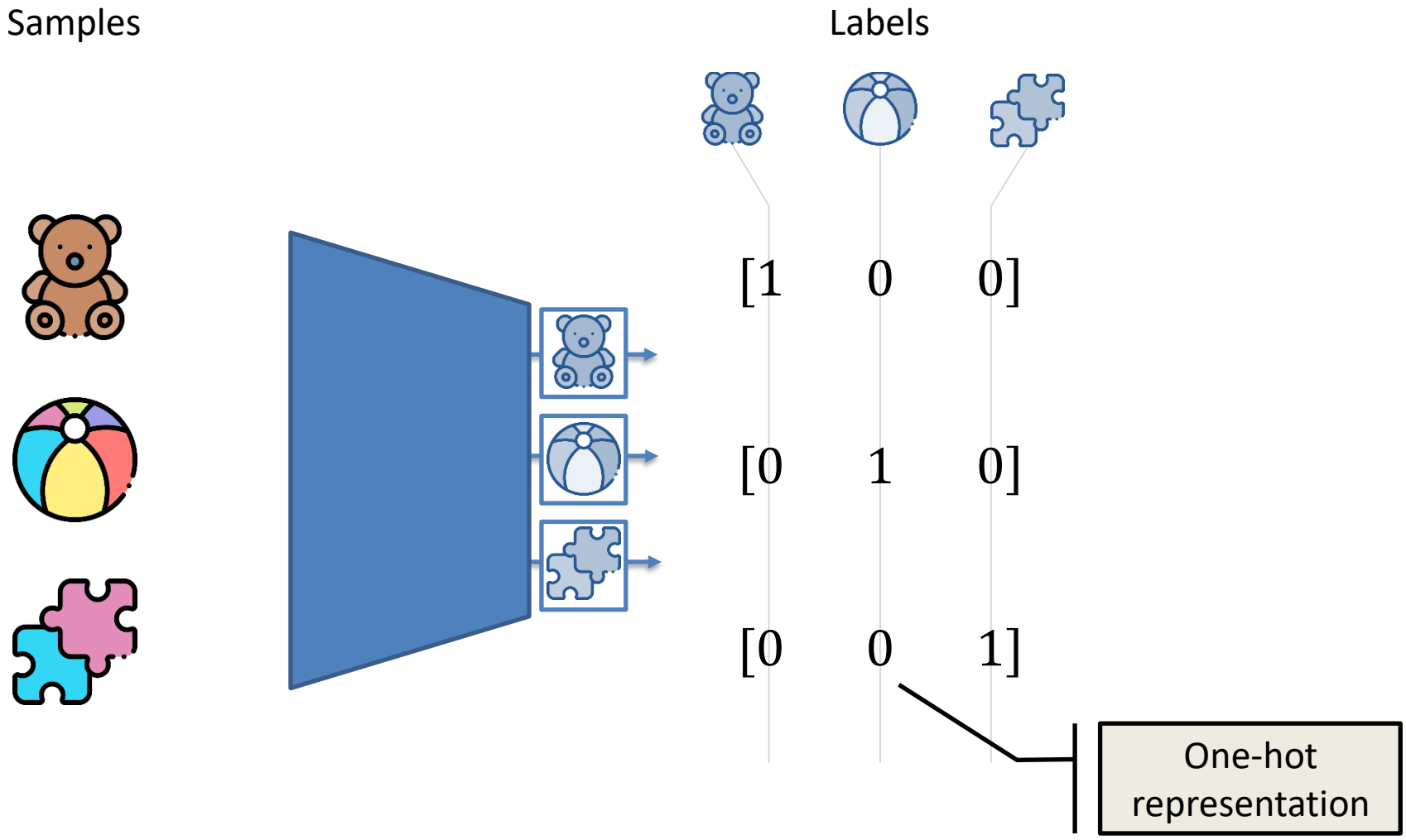
Supervised Learning Tasks



Tasks, Losses and the Output Layer

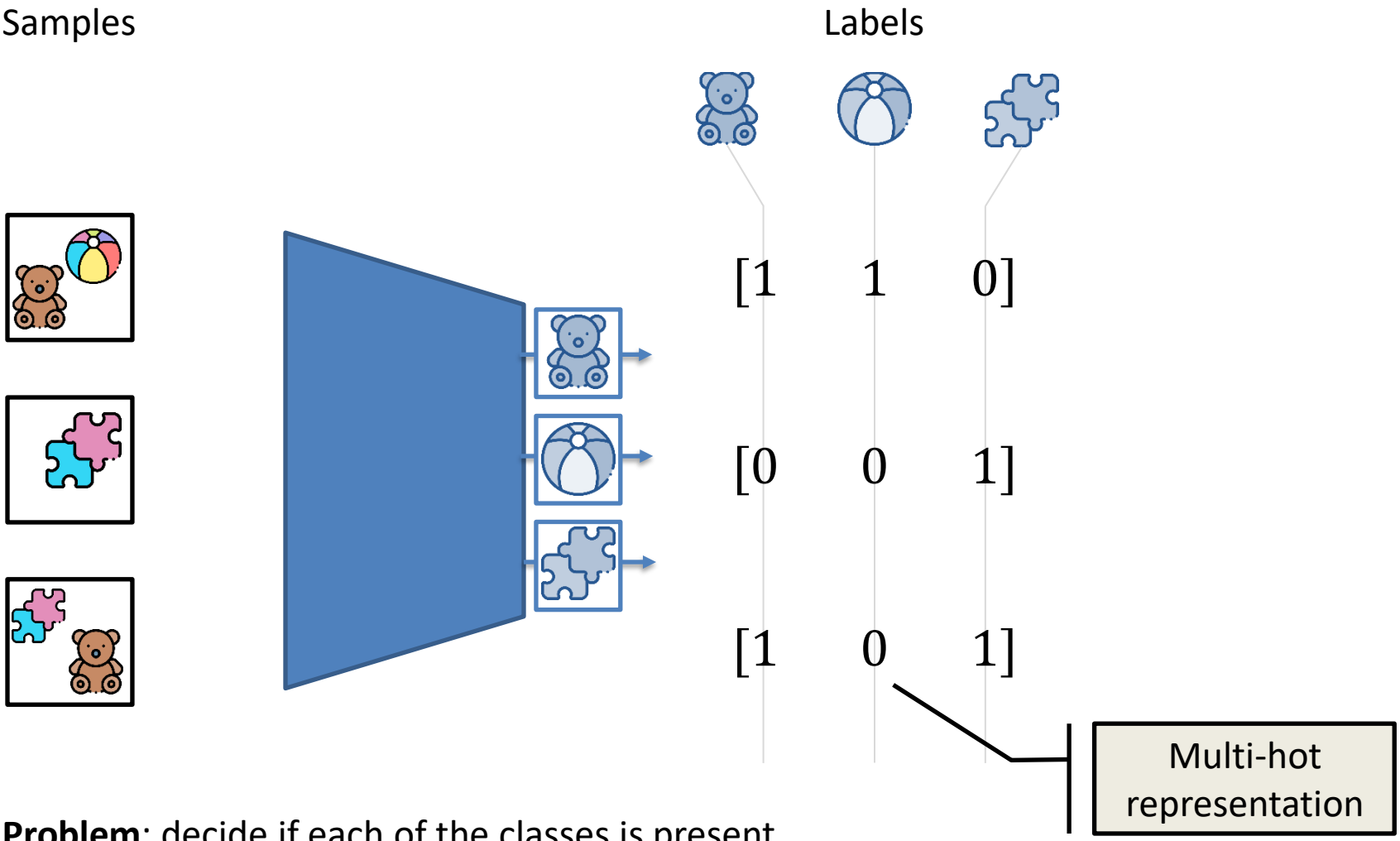
CLASSIFICATION

Multi-class classification



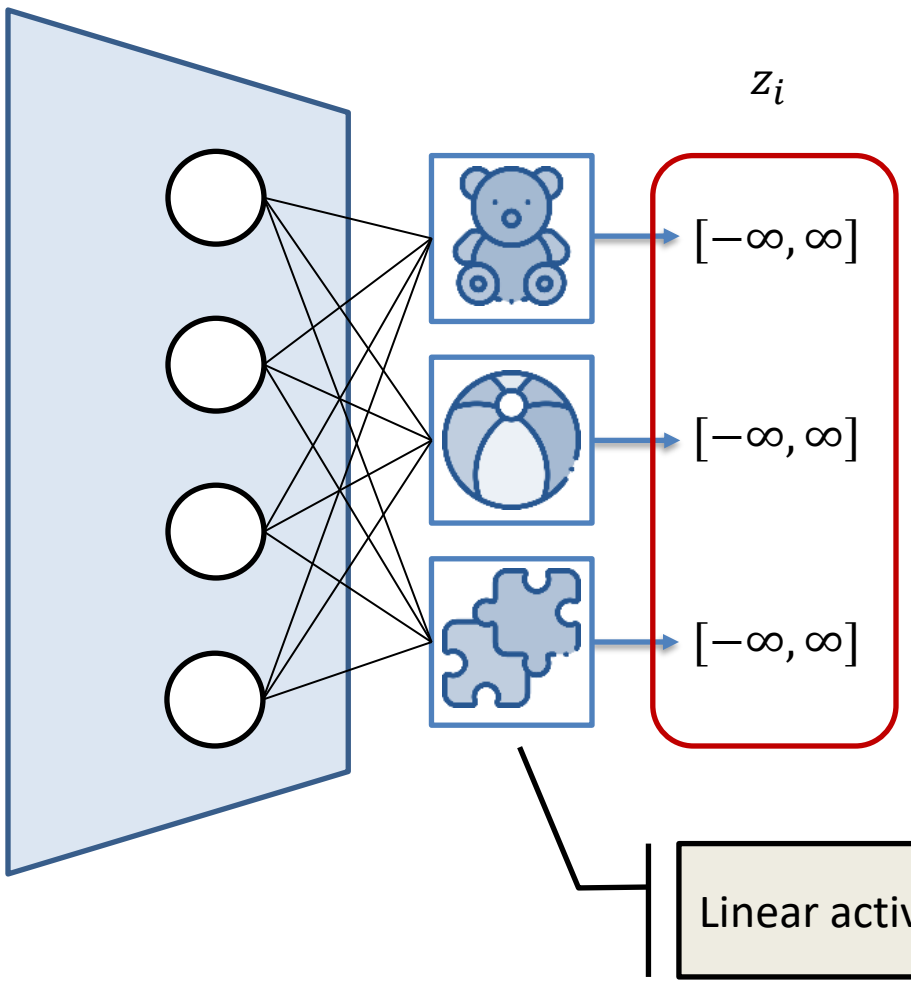
Problem: choose the best class, a **single classification problem**

Multi-label classification



Problem: decide if each of the classes is present,
N independent classification problems in parallel

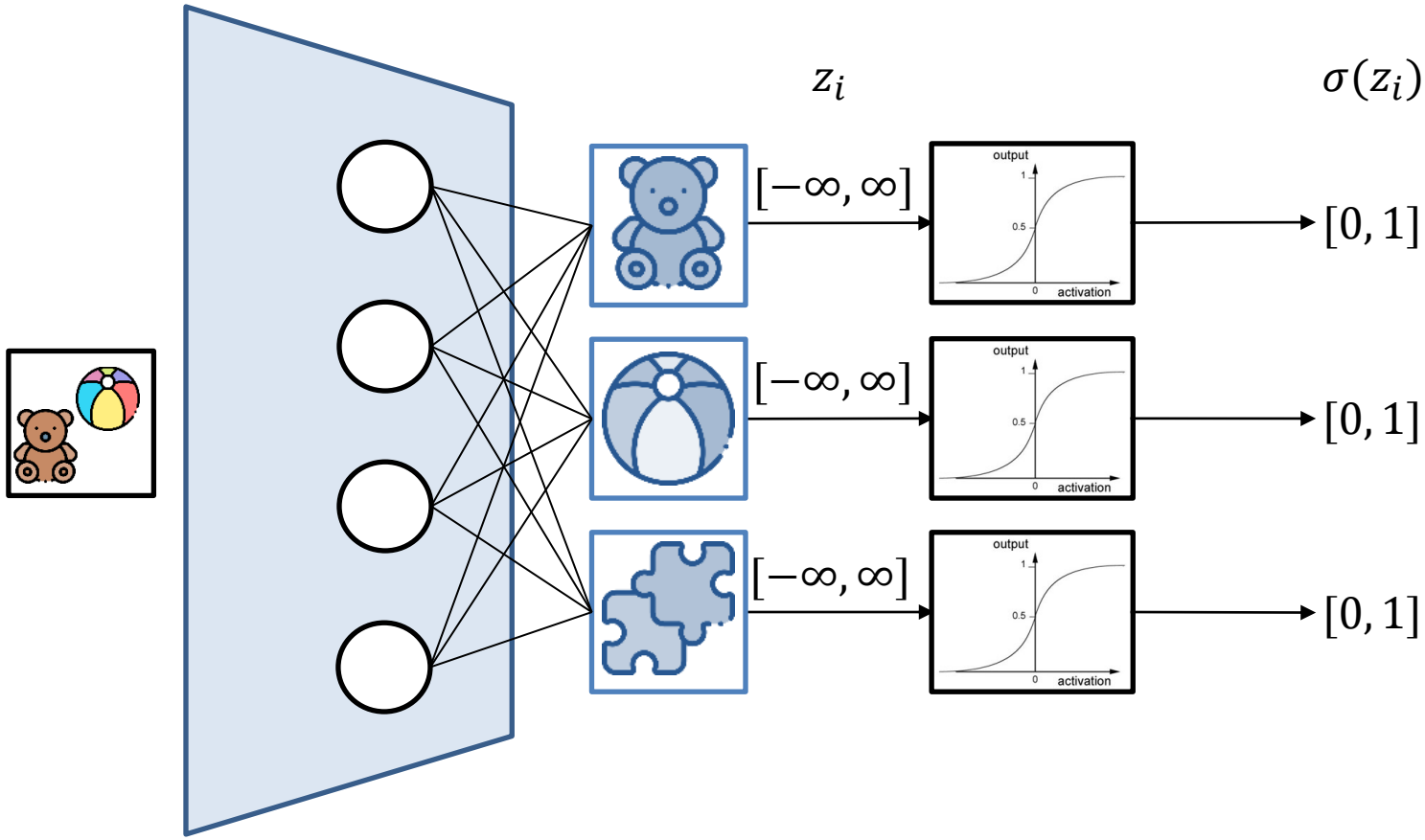
The output layer



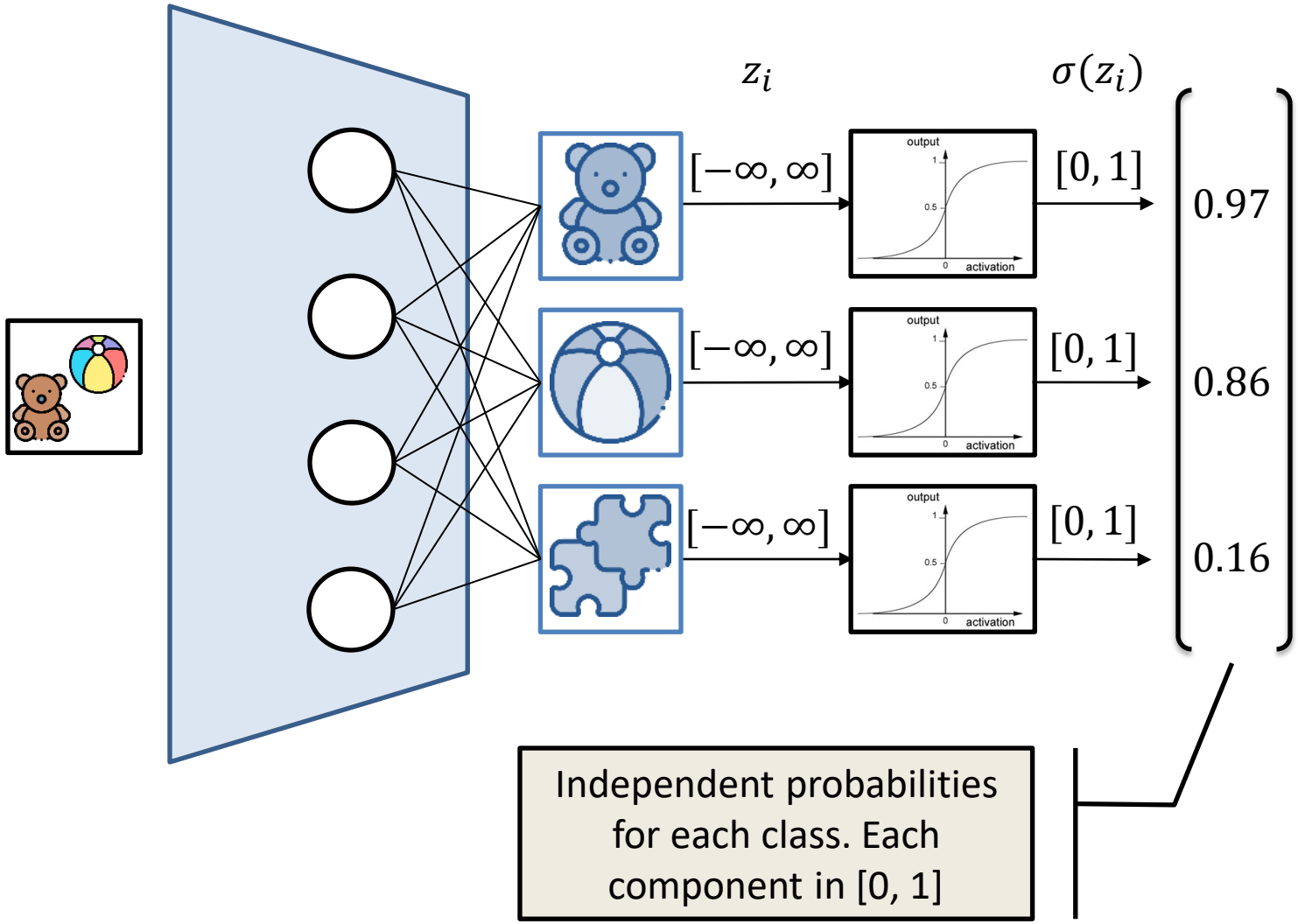
At this stage, a **linear activation** is probably the best choice:

- You need to reduce the output of the last layer to the **desired number of dimensions** (= number of classes)
- You want an output value which is **not bounded**, thinking about the classification (sigmoid) that will follow

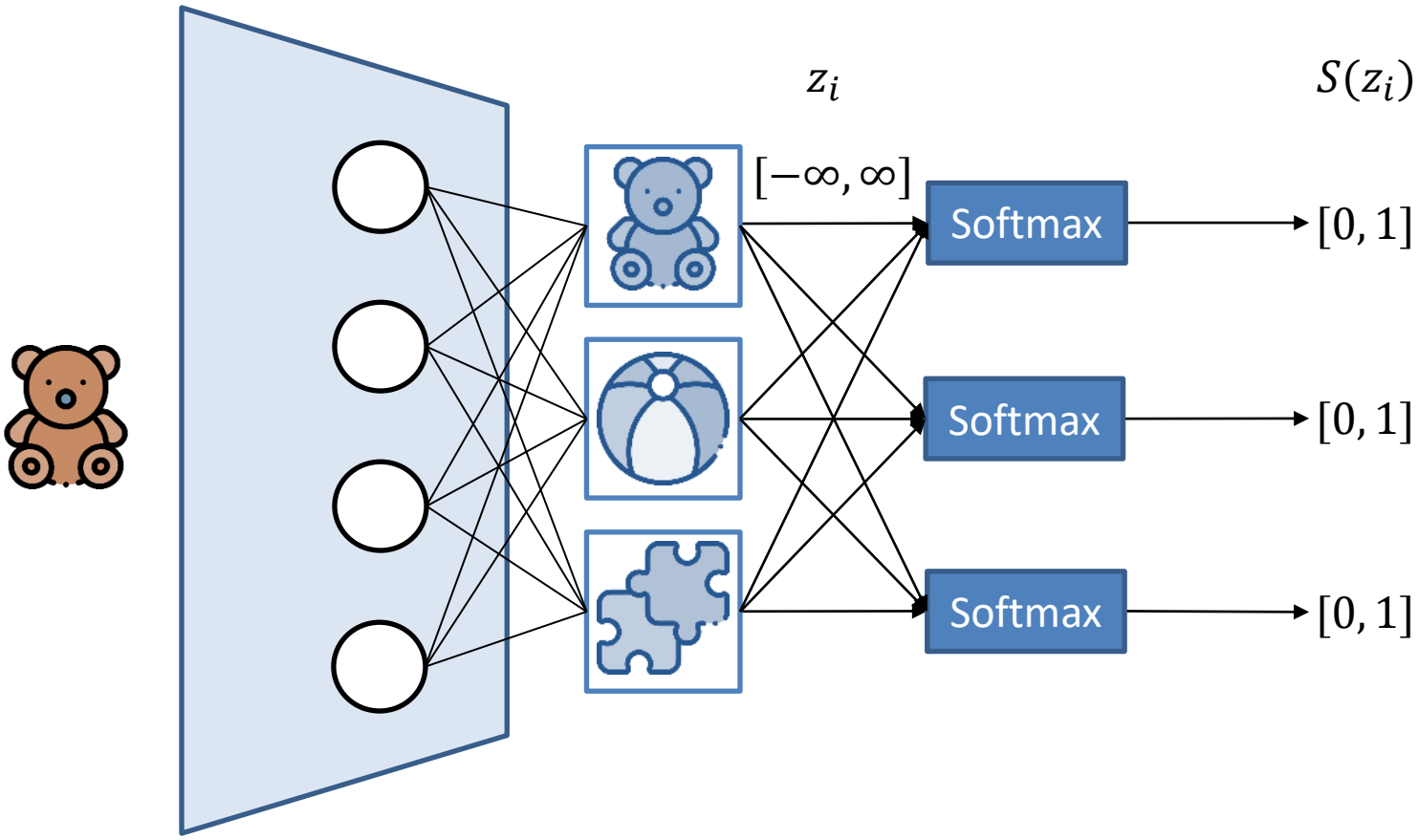
The output layer
Multi-label classification



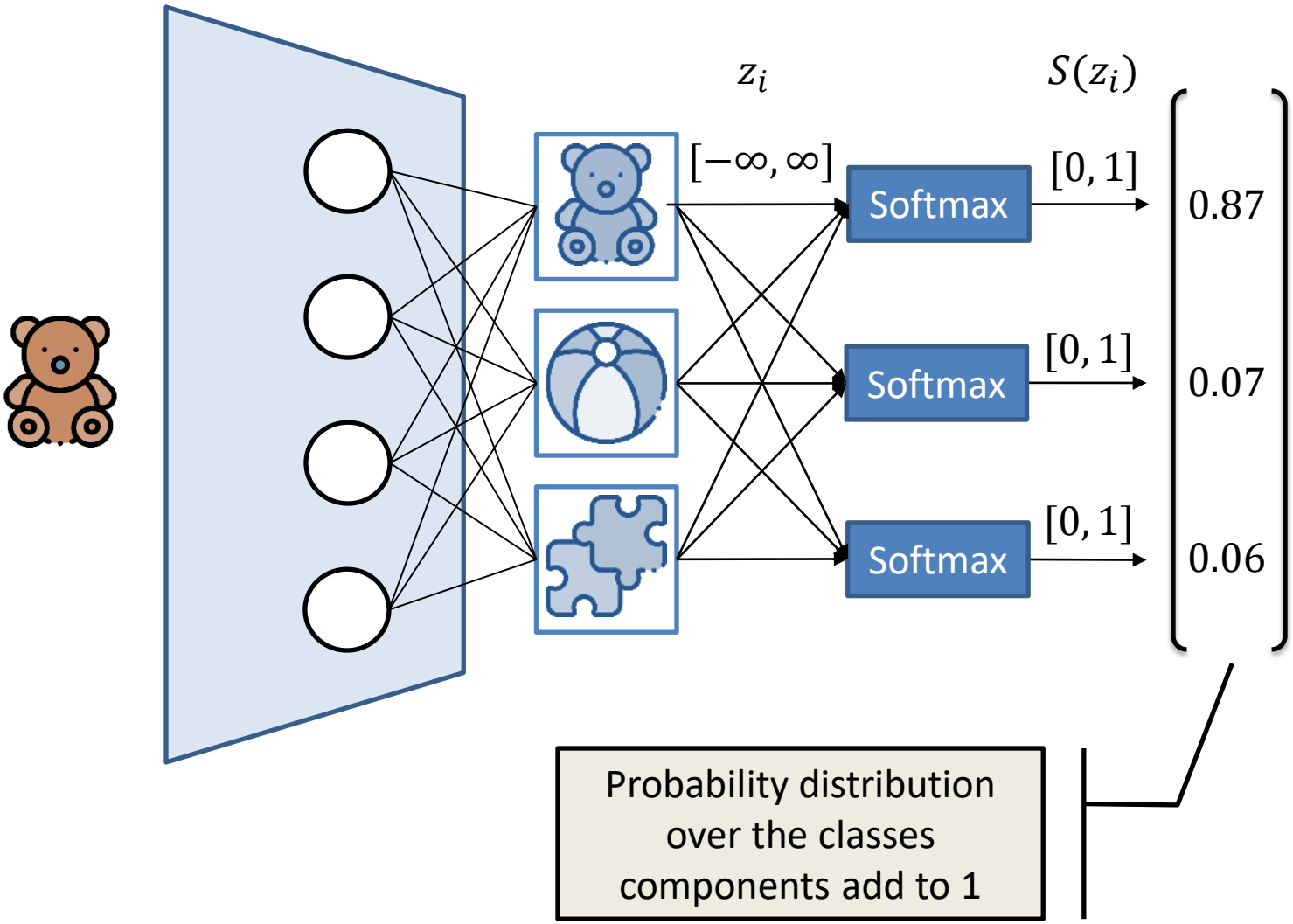
The output layer
Multi-label classification



The output layer
Multi-class classification



The output layer
Multi-class classification



Softmax

Softmax applies to a group of units (the logits). It is a generalization of the logistic function that squashes K arbitrary real values to K values in the range of [0, 1] that add up to one

$$S(z)_i = \frac{e^{z_i}}{\sum_j^C e^{z_j}}$$

$$\frac{\partial S_i}{\partial z_j} = S_i(\delta_{ij} - S_j)$$

$$\delta_{jj} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Try it out: http://neuralnetworksanddeeplearning.com/chap3.html#softmax

Cross-Entropy Loss

In both multi-label and multi-class scenarios, we will use different variants of the cross entropy loss to compare to the (multi-hot and one-hot) ground truth labels

The **cross entropy loss** for a class c is defined as:

$$J_{CE} = -\sum_{c} p_c \log \widehat{p_c}$$

$\widehat{p_c}$ is the estimated probability for class c

p_c is the true probability for class c. It is 1 if c is the correct class, otherwise it is 0

Binary Cross-Entropy Loss (for Multi-label classification)

The **binary cross-entropy loss** (that we saw in the case of logistic regression) is just a special case of the cross entropy loss

In the binary scenario with two classes c_1 and c_2 (or more commonly c and $\sim c$):

$$J_{BCE} = -\sum_c p_c \log \widehat{p_c} = -p_{C_1} \log(\widehat{p_{c_1}}) - p_{C_2} \log(\widehat{p_{c_2}})$$

$$p_{C_1} + p_{C_2} = 1$$

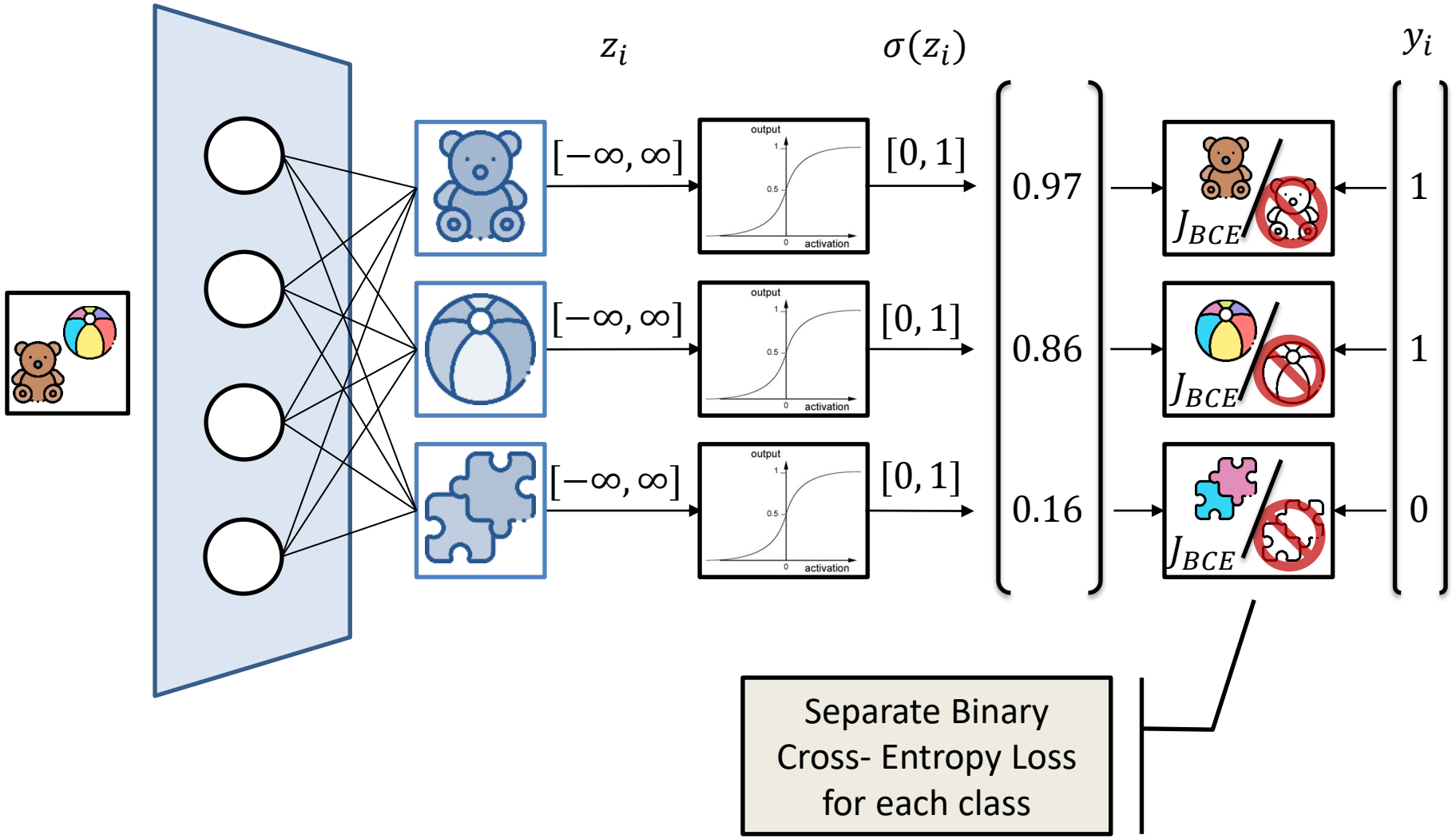
$$\hat{p}_{C_1} + \hat{p}_{C_2} = 1$$

$$J_{BCE} = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

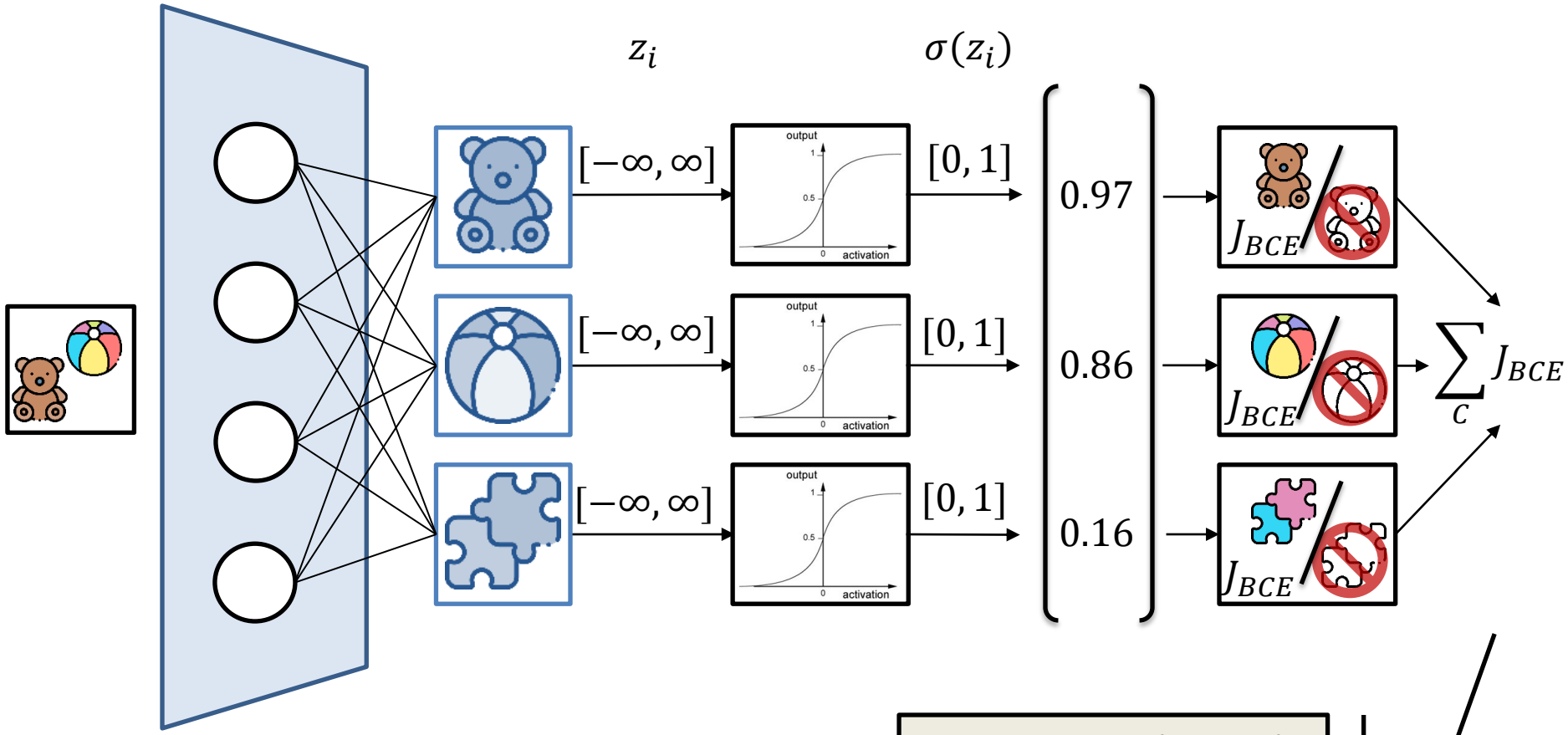
$\hat{y}$ is the estimated probability for class c_1 in [0, 1]

y is the true label, it can only be 1 or 0

The output layer
Multi-label classification

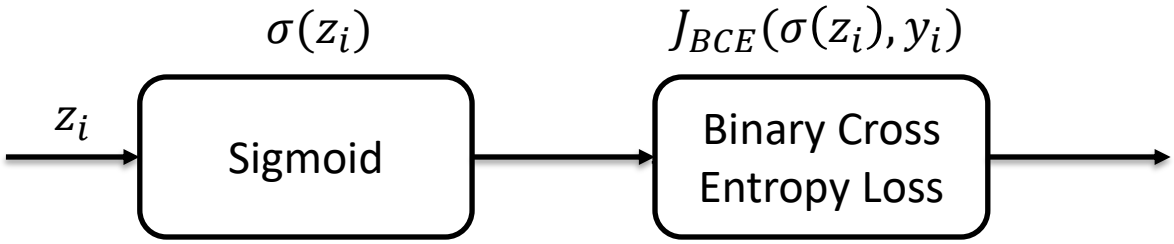


The output layer
Multi-label classification



To compute the total loss we simply sum up the individual losses

Derivatives – Binary Cross Entropy Loss

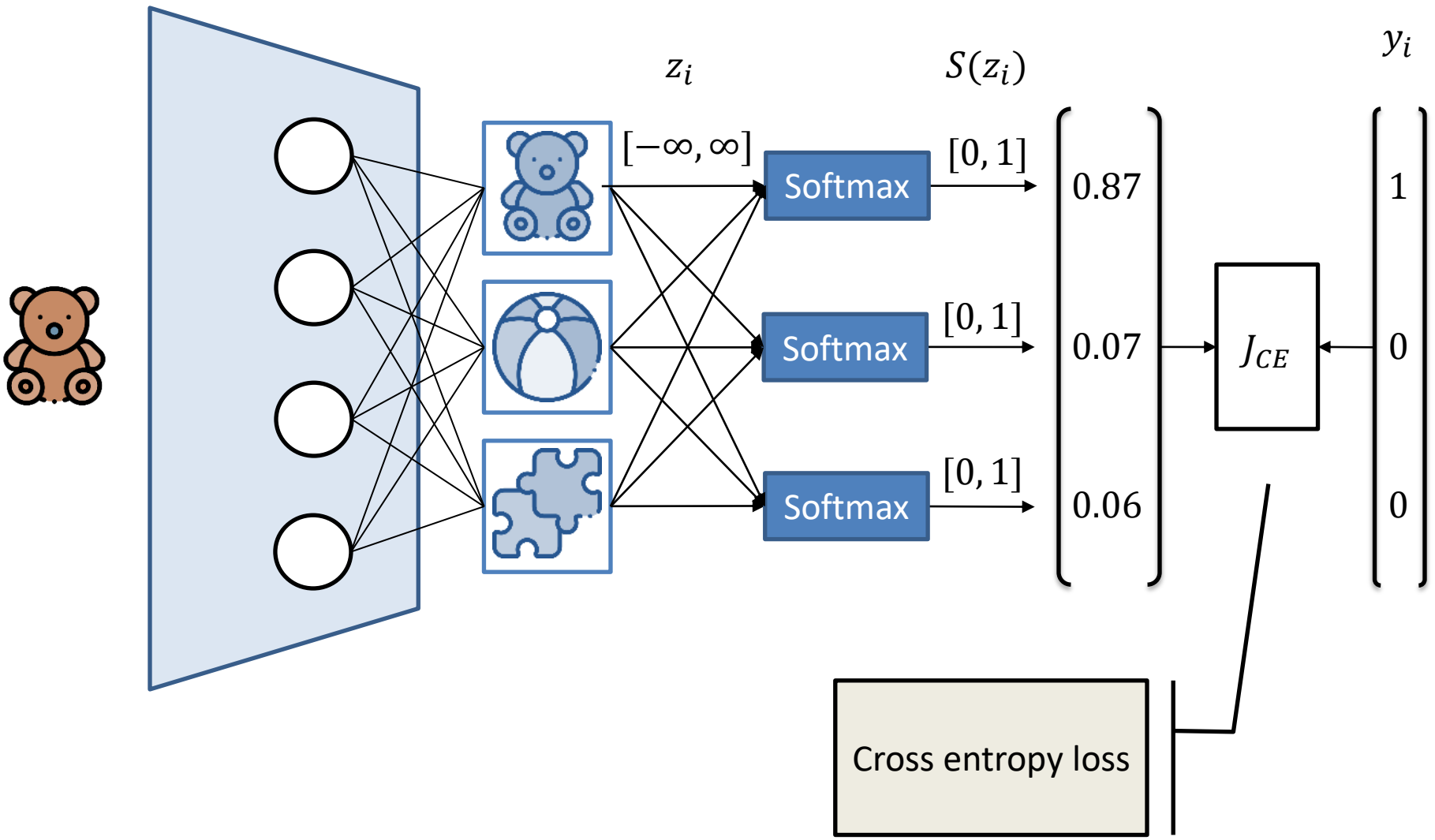


$$\frac{\partial J_{BCE}}{\partial z_i} = \begin{cases} \sigma(z_i) - 1 & \text{if } y_i = 1 \\ \sigma(z_i) & \text{if } y_i = 0 \end{cases}$$

Check the Docs:

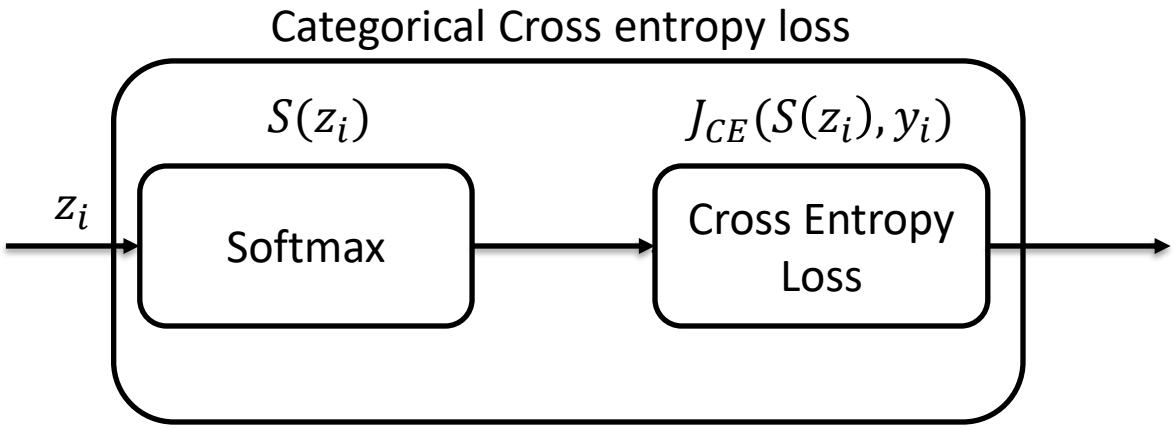
Pytorch: BCELoss or BCEWithLogitsLoss
TensorFlow: BinaryCrossentropy
Caffe: Multinomial Logistic Loss Layer

The output layer
Multi-class classification



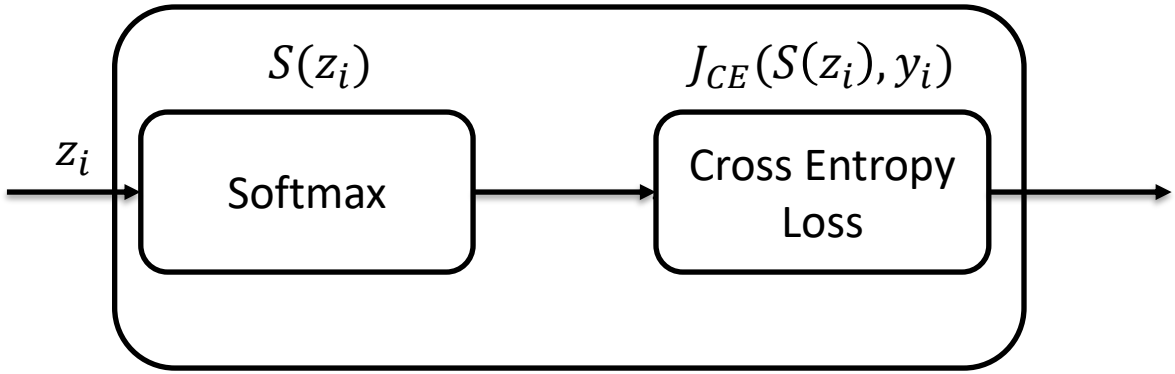
Categorical Cross Entropy Loss (for Multi-class classification)

The combination of a Softmax followed by cross entropy loss is called **Categorical Cross Entropy Loss** or simply **Softmax Loss**



It turns out it is computationally more efficient to treat these two steps in a single go

Categorical Cross Entropy Loss (for Multi-class classification)

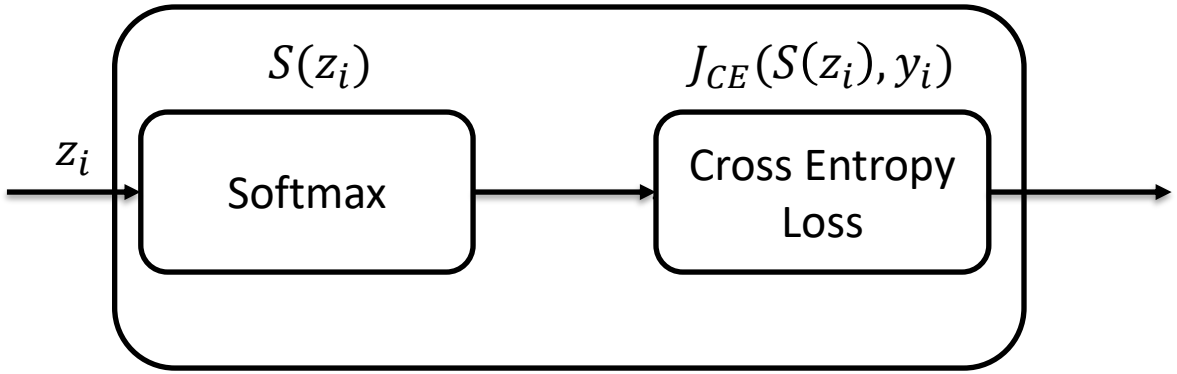


$$S(z)_i = \frac{e^{z_i}}{\sum_j^C e^{z_j}}$$

$$J_{CE} = -\sum_i^C y_c \log(S(z)_i)$$

The positive class is the only one keeping its term in the loss

Categorical Cross Entropy Loss (for Multi-class classification)



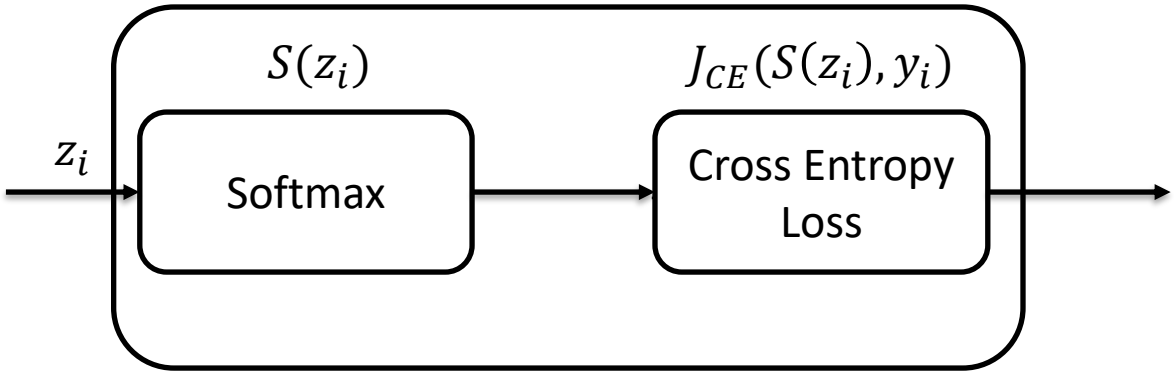
$$S(z)_i = \frac{e^{z_i}}{\sum_j^C e^{z_j}}$$

$$J_{CCE} = -\sum_i^C y_c \log(S(z)_i)$$

The logit of the positive class

$$J_{CCE} = -\log\left(\frac{e^{z_p}}{\sum_j^C e^{z_j}}\right)$$

Derivatives – Categorical Cross Entropy Loss



$$\frac{\partial J_{CCE}}{\partial z_i} = \begin{cases} S(z_i) - 1 & \text{if } y_i = 1 \\ S(z_i) & \text{if } y_i = 0 \end{cases}$$

Check the Docs:

Pytorch: CrossEntropyLoss (or LogSoftmax and NLLLoss)
TensorFlow: CategoricalCrossentropy
Caffe: SoftmaxWithLoss Layer

Softmax – what's in the name

Softmax() is not actually a soft version of the *max()* function, but of the *argmax()* function. You can see this by introducing a multiplier α to all inputs

$$S(\alpha z)_i = \frac{e^{\alpha z_i}}{\sum_j^C e^{\alpha z_j}}$$

As $\alpha \to \infty$, the maximum term in the inputs will dominate the sum, and the *Softmax()* will approximate the *argmax()*

A soft version of the *max()* function would rather be given by $S(z)^T z$

SoftMax - numerical stability

Softmax() can be numerically unstable, as it exponentiates the inputs and ends up calculating divisions of very large numbers

Note that *Softmax()* only cares about the differences between the inputs, not their real values. You can easily see that if you add a constant β to all the inputs it does not affect the result:

$$S(z+\beta)_i = \frac{e^{z_i+\beta}}{\sum_j^C e^{z_j+\beta}} = \frac{e^{\beta}e^{z_i}}{e^{\beta}\sum_j^C e^{z_j}} = \frac{e^{z_i}}{\sum_j^C e^{z_j}} = S(z)_i$$

SoftMax - numerical stability

This gives us a neat way to improve the numerical stability of Softmax by avoiding divisions between very large numbers due to the exponentials. Just shift all the input values so that the highest value is zero, before you pass them through Softmax

```
# Activation function
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x)) # Not safe, potential numerical problems

def stable_softmax(x):
    x -= np.max(x) # shift all input values that the highest number is 0
    return np.exp(x) / np.sum(np.exp(x)) # safe to do, gives the correct answer
```

```
out = softmax([20, 134, 854])
print(out)
```

```
[ 0.  0. nan]
```

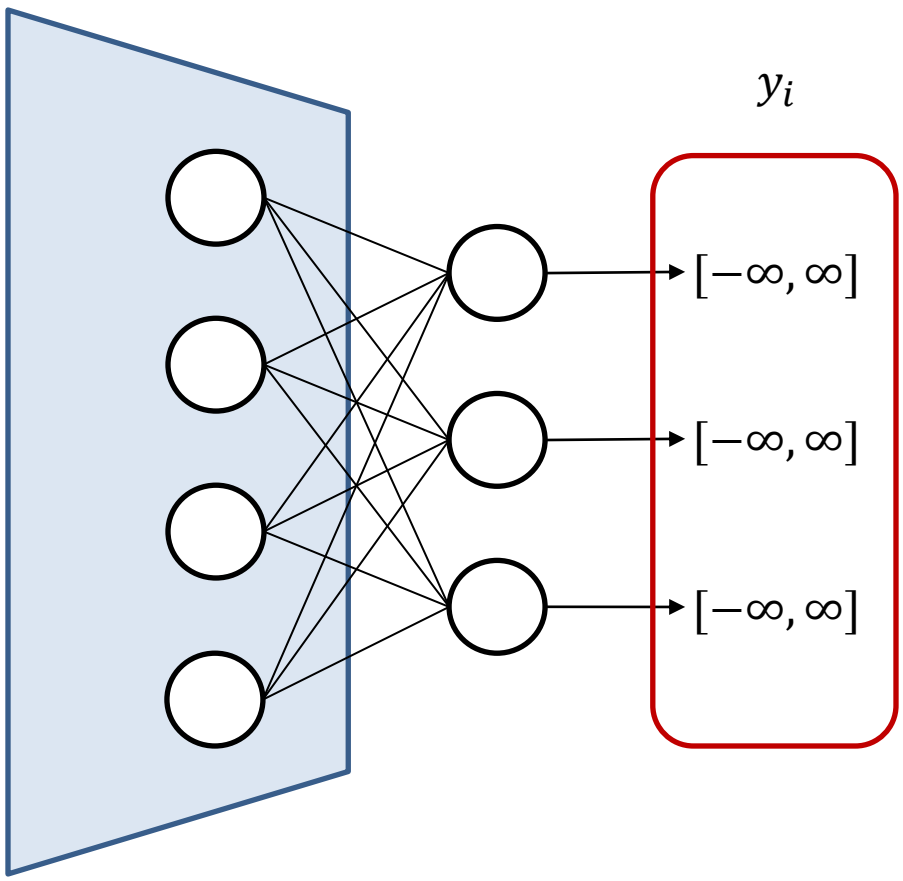
```
out = stable_softmax([20, 134, 854])
print(out)
```

```
[0.0000000e+000 2.0322308e-313 1.0000000e+000]
```

Tasks, Losses and the Output Layer

REGRESSION

The output layer



At this stage, a **linear activation** is also a good choice:

- Bring the output to the **desired number of dimensions**
- You want an output value which in a desired range (typically unbounded)

Loss functions for regression

Mean Square Error (MSE)

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Mean Absolute Error (MAE)

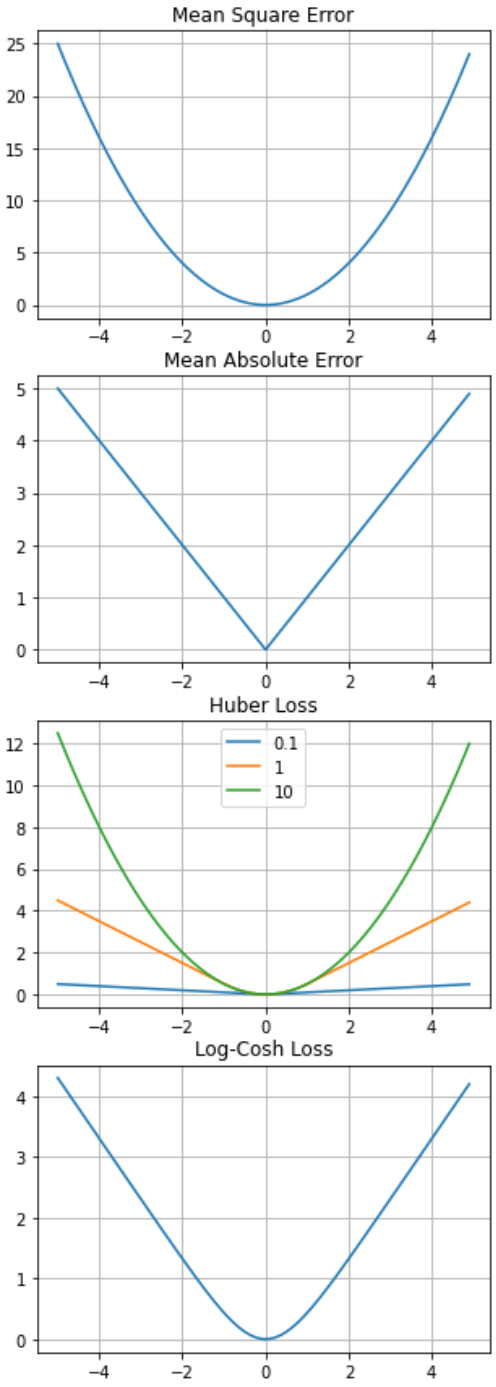
$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|$$

Huber Loss (smooth MAE)

$$L_\delta = \frac{1}{m}\sum_{i=1}^{m}\begin{cases}\frac{1}{2}(y_i - \hat{y}_i)^2 & |y_i - \hat{y}_i| < \delta \\ \delta|y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{otherwise}\end{cases}$$

Log-Cosh Loss

$$L_{lc} = \frac{1}{m}\sum_{i=1}^{m}\log(\cosh(\hat{y}_i - y_i))$$



Regression vs Classification

Note that regression problems using the MSE loss tend to be much harder to optimise than classification problems using a more stable loss such as *Softmax*

Mean Square Error

- Aims to predict the exact correct value
- Outliers (large distances) cause large gradients

Softmax

- The exact values do not matter, as long as the relative magnitudes are correct
- Outliers saturate the loss, less critical

Consider whether a regression problem can be quantised and posed as a classification one

Check the PyTorch Documentation:

https://pytorch.org/docs/stable/nn.html#loss-functions

NEURAL NETWORK ARCHITECTURES

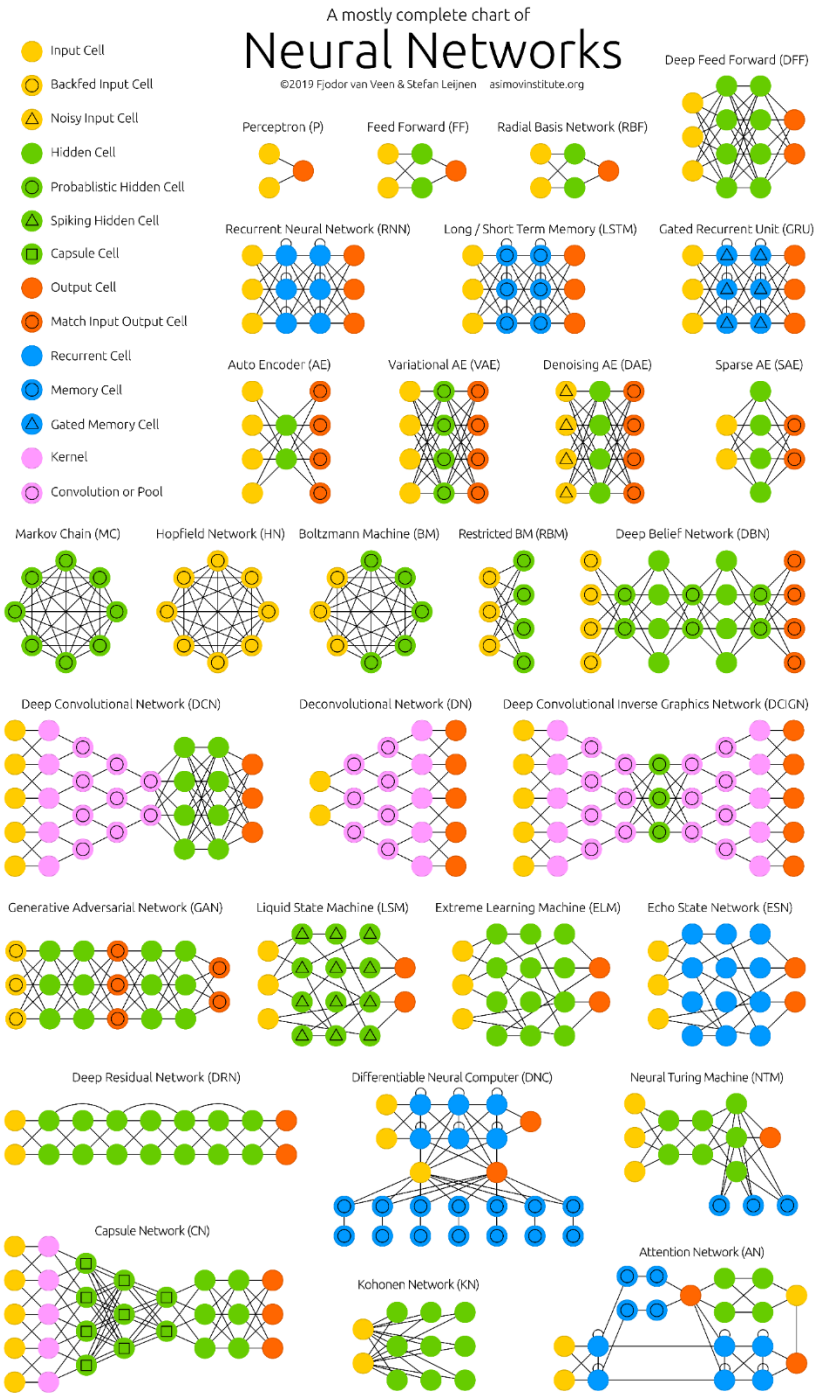
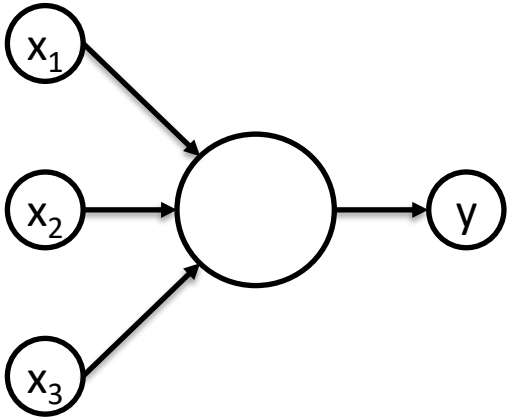


Image from:
https://www.asimovinstitute.org/neural-network-zoo/

Single Neuron (Perceptron)

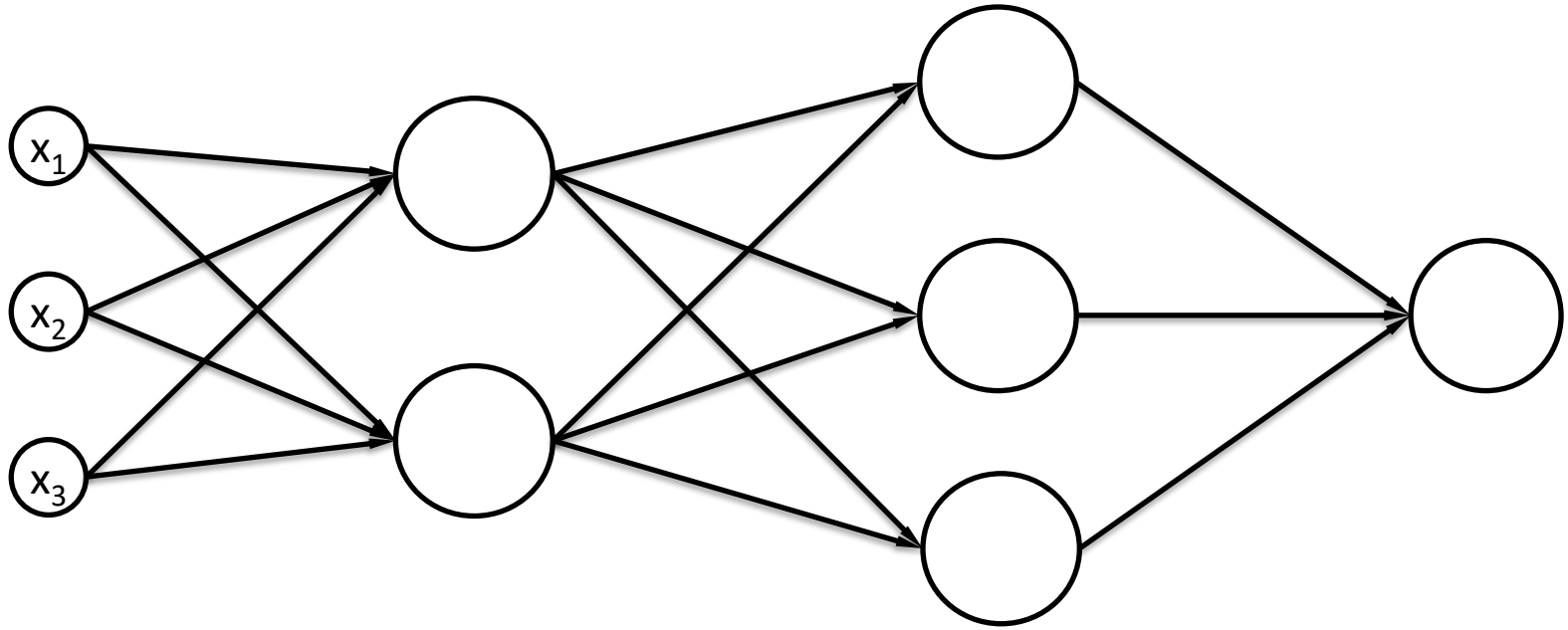
With a single neuron we can basically do linear / logistic regression (depending on how we define the activation function)



$$y = h\left(b + \sum_i w_i x_i\right)$$

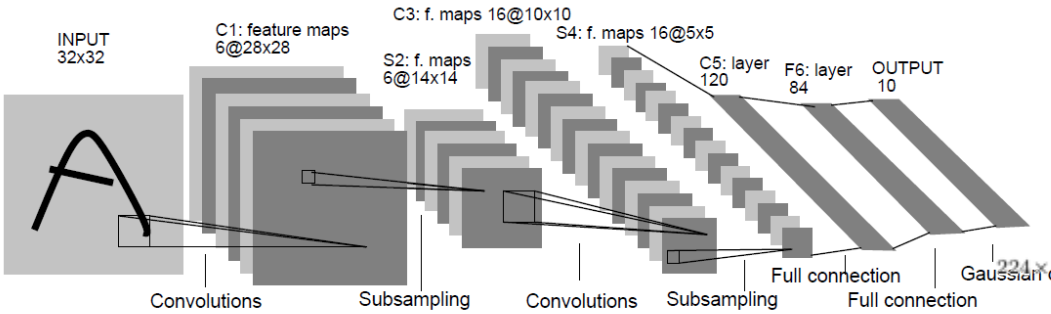
Feed-Forward Neural Networks

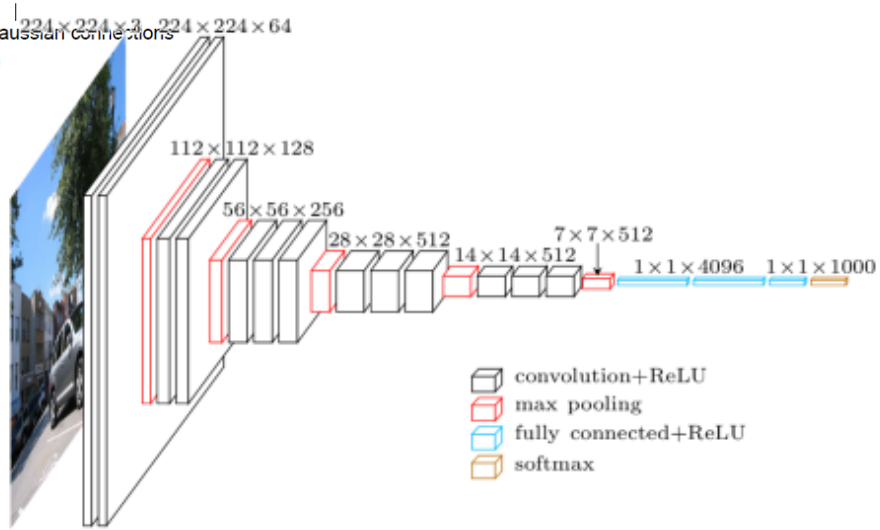
No-cycles, all connections point to the same direction
They perform a series of transformations of the input data (that change the similarities between cases) before the final output is calculated
If a feed forward NN has more than one hidden layer, it is called “deep”



Convolutional Neural Networks

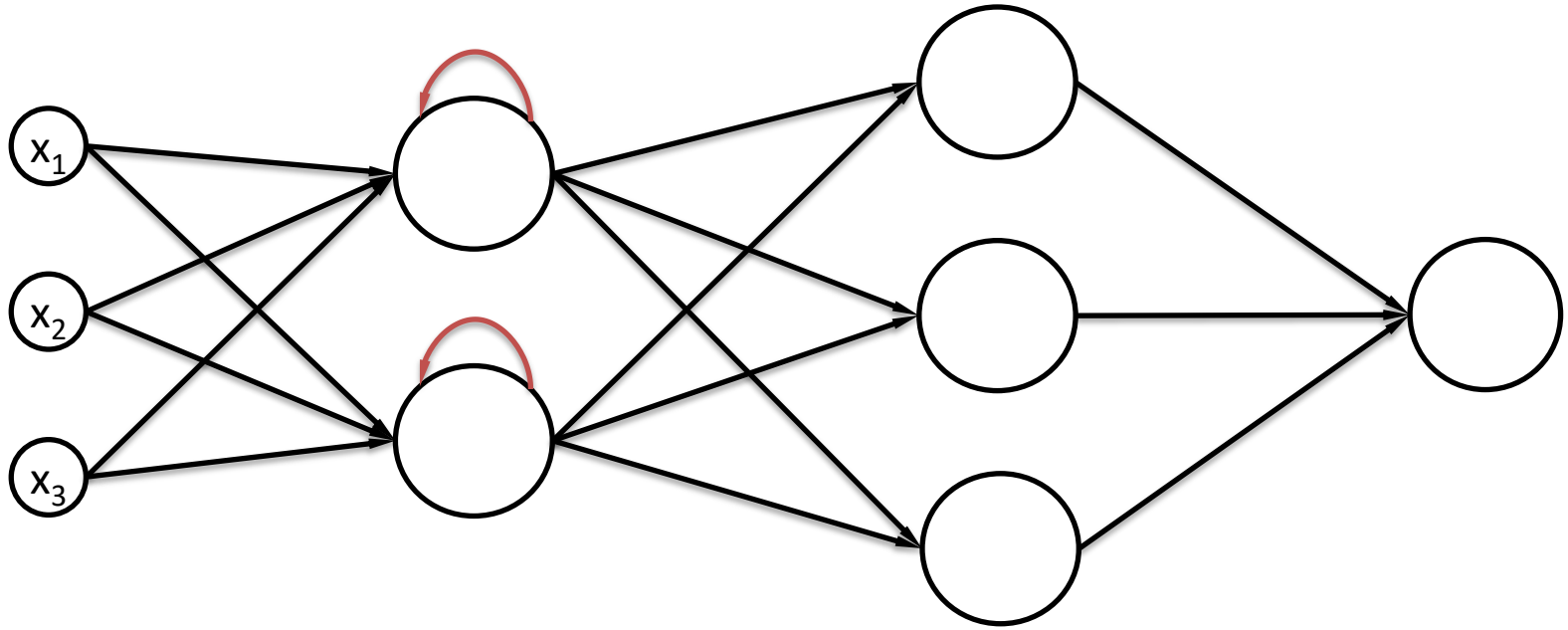
Convolutional neural networks extract **replicated features** from the input signal (local receptive fields and shared weights). Unlike fully connected networks, convolutional networks take into account the spatial structure of an image.





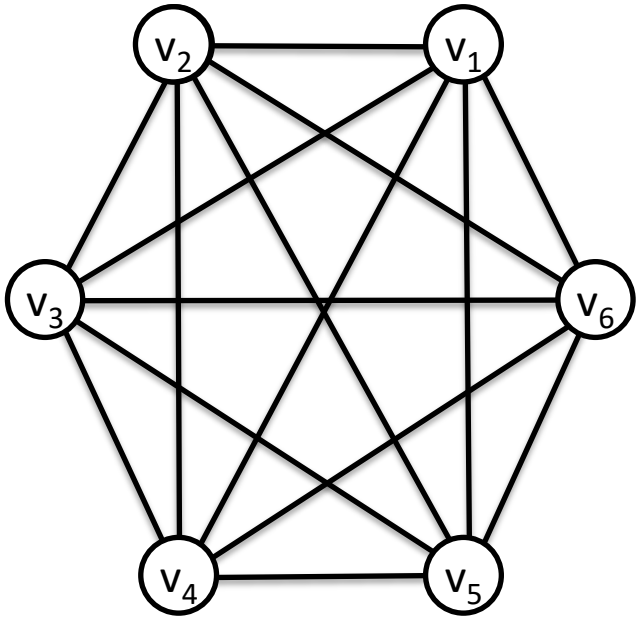
Recurrent Neural Networks

If we allow cycles, we get recurrent neural networks. RNNs have “states”. They offer a natural way to model sequential data.



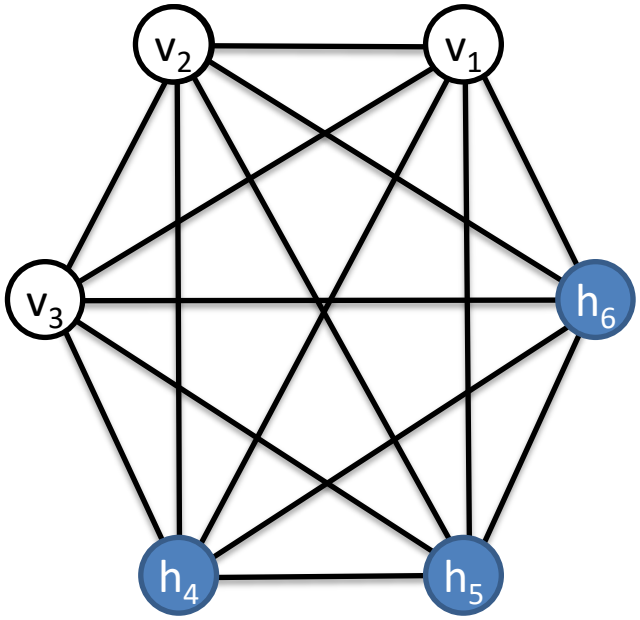
Hopfield Network

Similar to recurrent networks, but the connections are symmetrical (they have the same weight in both directions)



Symmetrically Connected Nets with hidden units (Boltzmann Machines)

The stochastic counterpart of Hopfield nets. Much more powerful models with a simple learning algorithm.



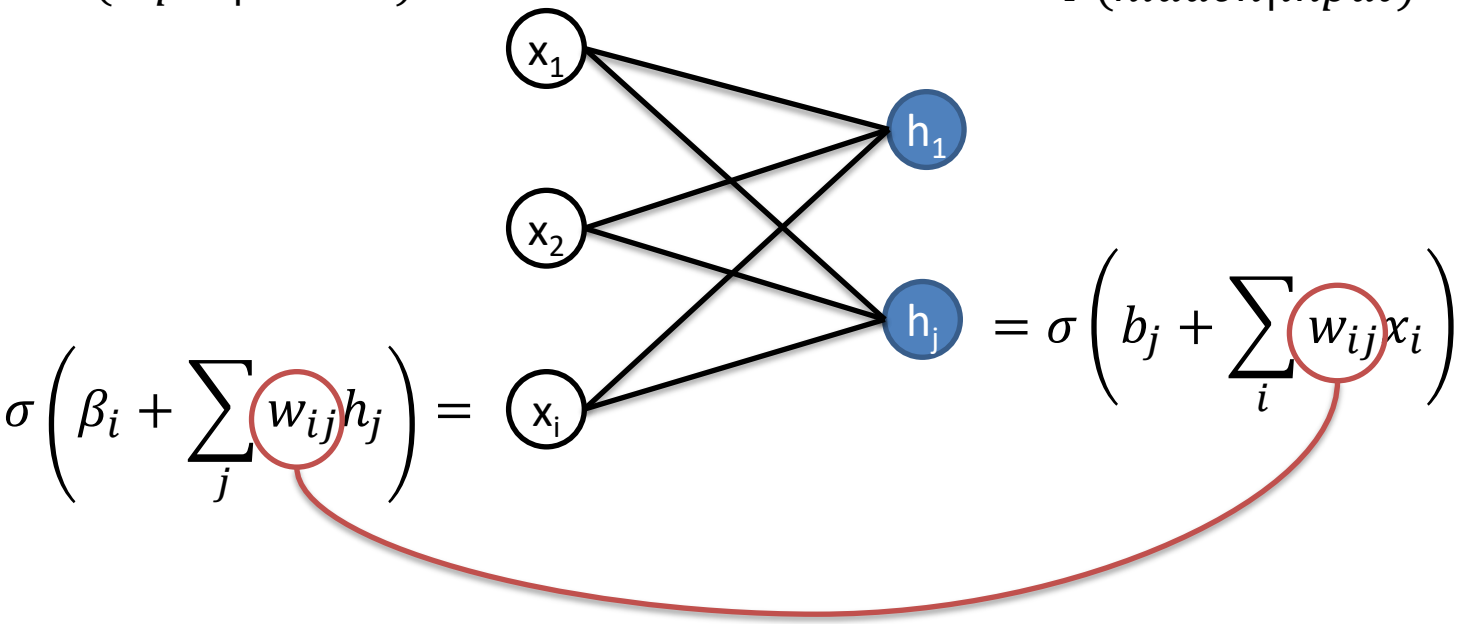
Restricted Boltzmann Machines

Like Boltzmann machines, but their neurons must form a bipartite graph: there are no connections between nodes within a group.

Trained to maximise the likelihood of input data. It is the equivalent to calculating a mixture model.

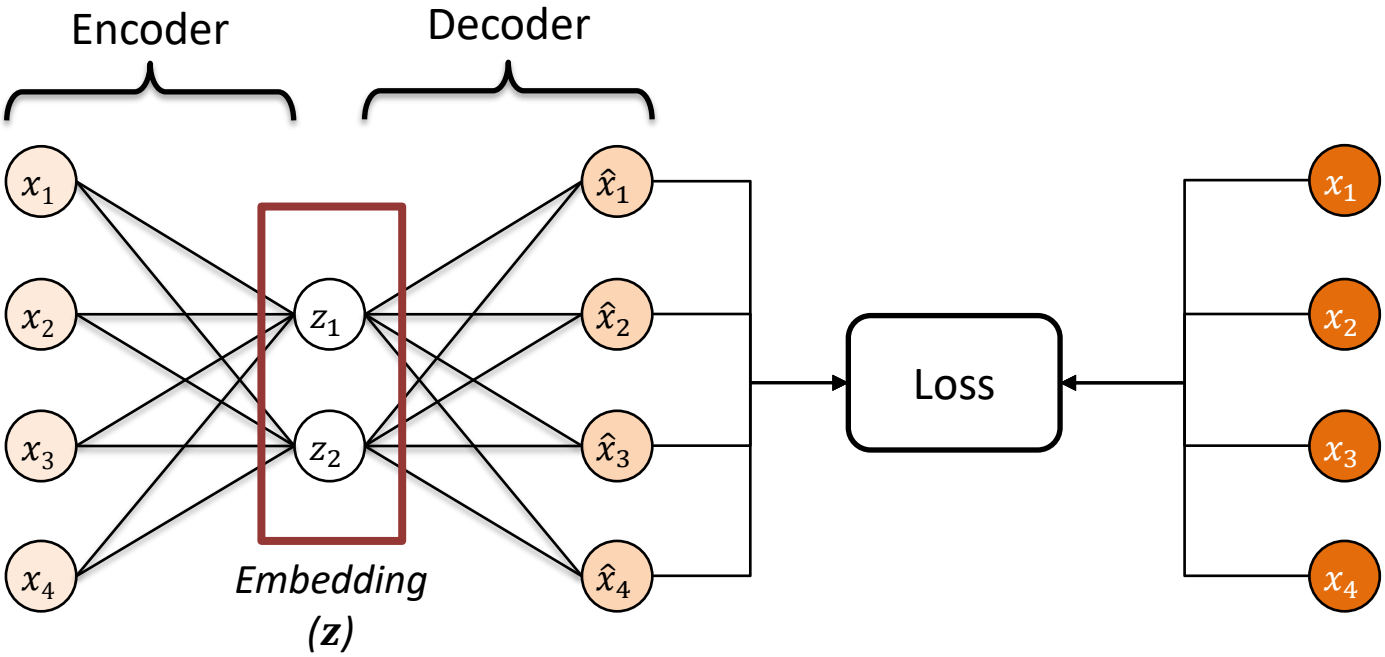
$P(input|hidden)$

$P(hidden|input)$



$$\sigma\left(\beta_i + \sum_j w_{ij} h_j\right) = \sigma\left(b_j + \sum_i w_{ij} x_i\right)$$

Autoencoders

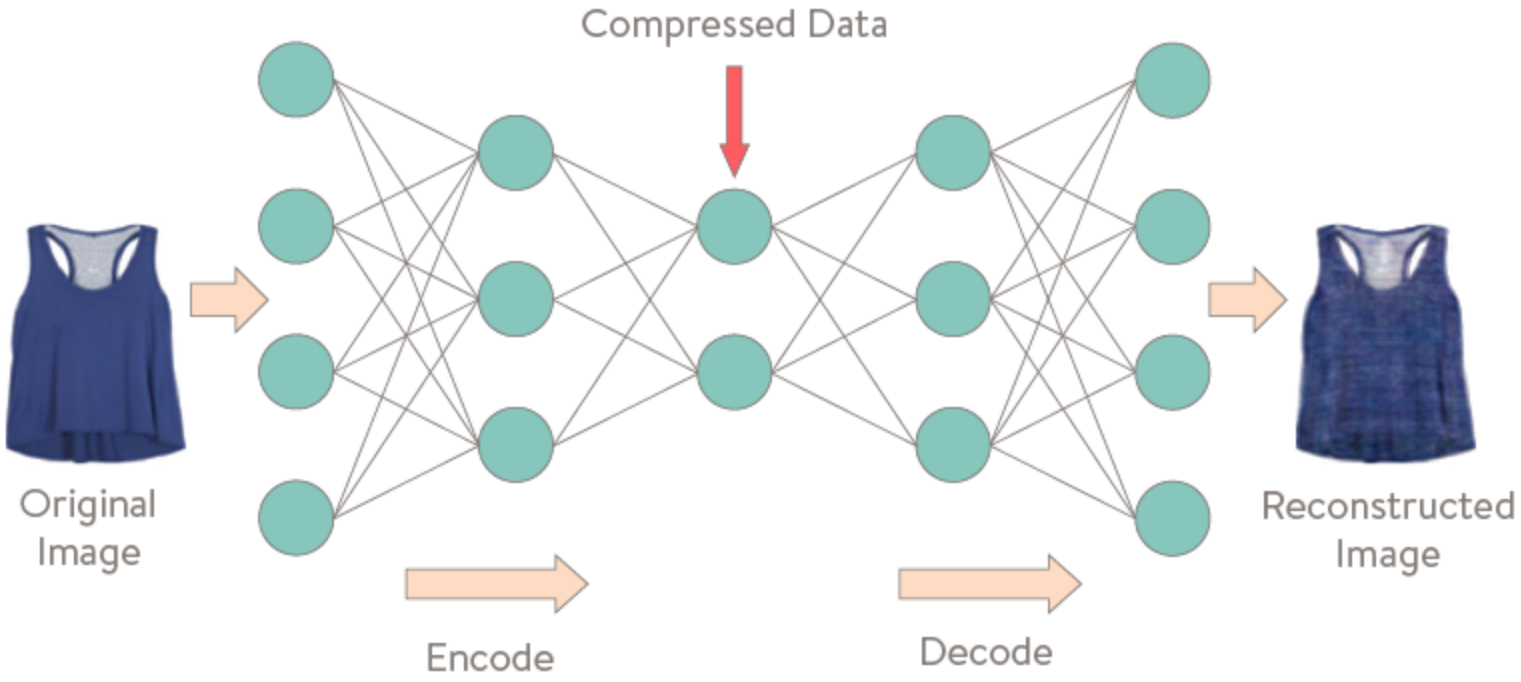


Given a training set without annotations: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \ldots$

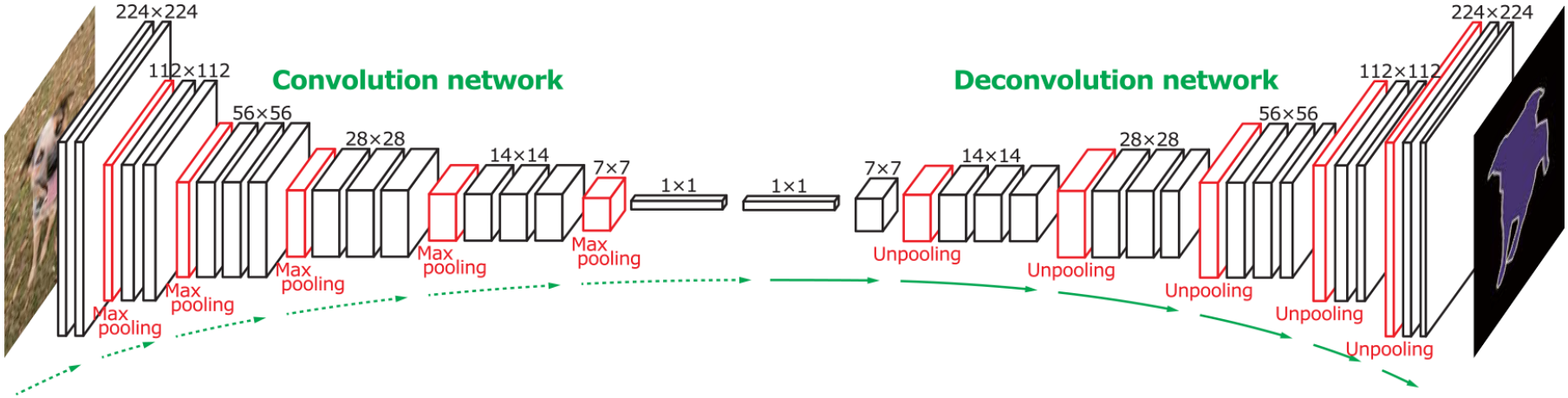
Adjust parameters $\mathbf{W}$ (of every layer) to make: $f_{\mathbf{W}}\left(\mathbf{x}^{(i)}\right) \approx \mathbf{x}^{(i)}$

Learn useful (usually compact) representations ($\mathbf{z}$)

Autoencoders



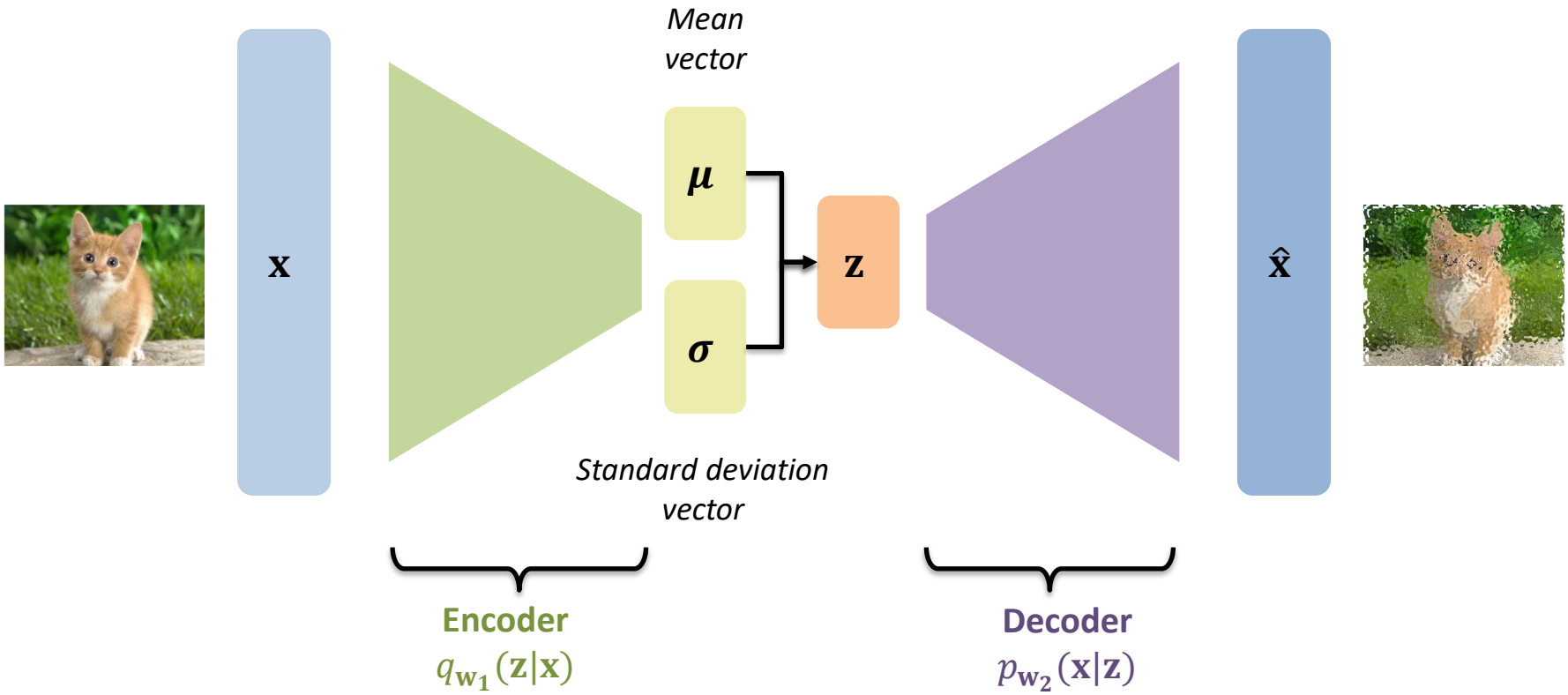
Fully Convolutional Networks



Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015.

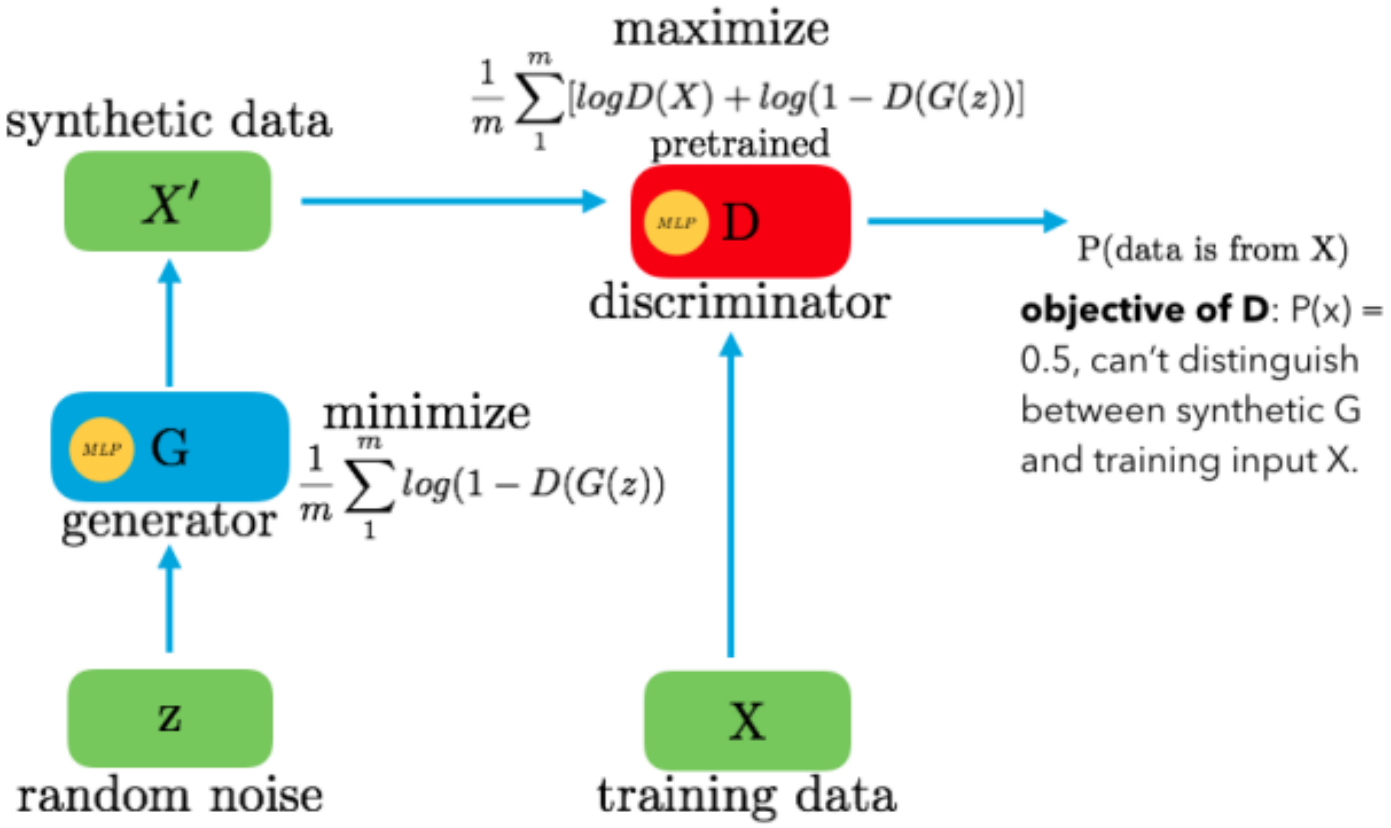
Variational Autoencoders (VAE)

Input → encode to statistics vectors → sample a latent vector → decode for reconstruction

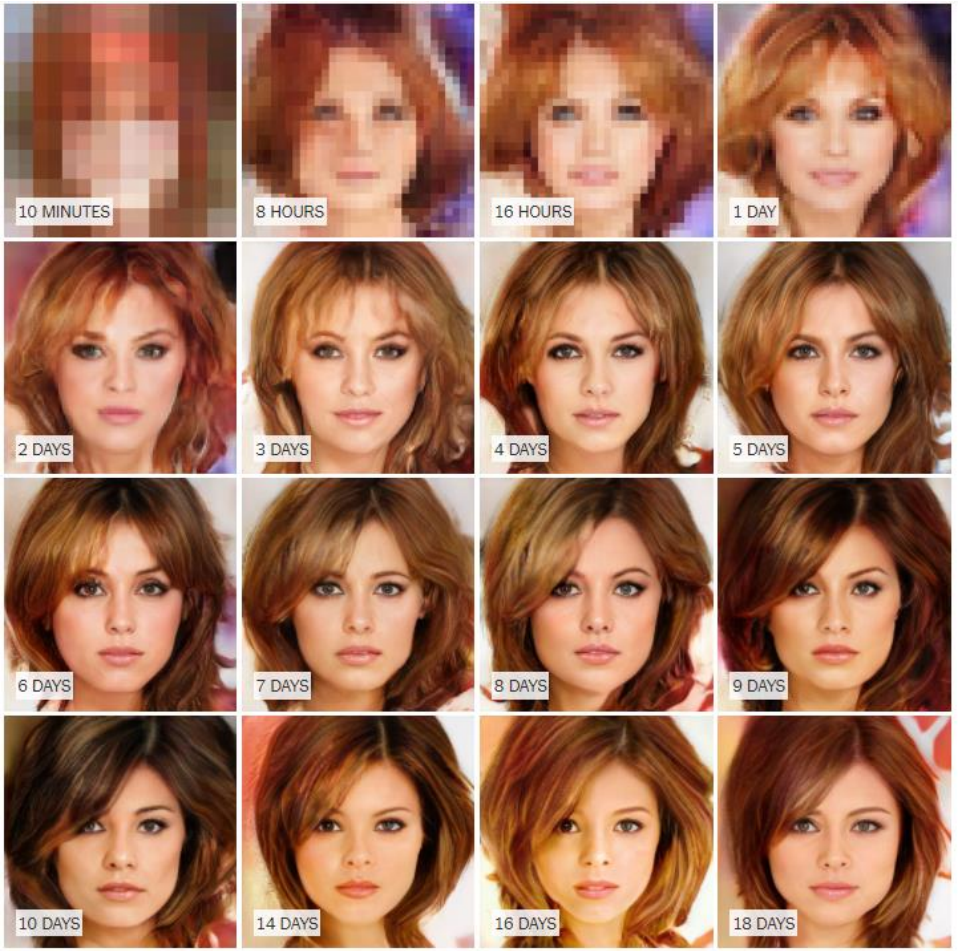


$$L(w_1, w_2, x) = (reconstruction\ loss) + (regularization\ term)$$

Generative Adversarial Networks



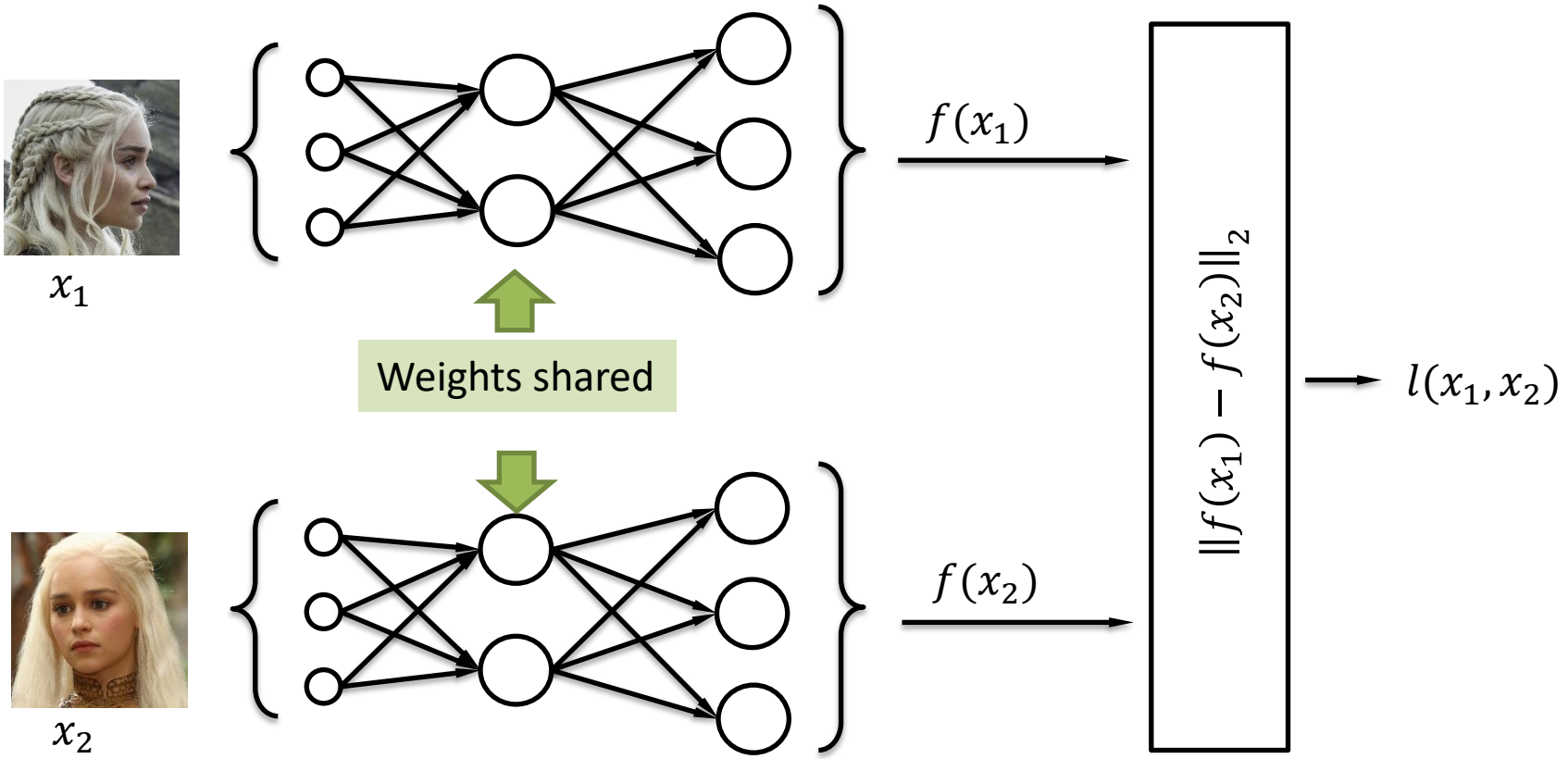
Real or Fake?



https://www.nytimes.com/interactive/2018/01/02/technology/ai-generated-photos.html

Siamese networks

Two parallel feed-forward networks, with **shared** weights. Produce a representation of the input sample. We adjust the weights so that the distance of the representations of same things is small and the distance of the produced representations for dissimilar things is large. Used for metric learning.



Bromley, J. et al, 1993. Signature Verification Using A "Siamese" Time Delay Neural Network. IJPRAI, 7(4), pp.669-688.

Triplet loss

Instead, we could adjust the weights so that the distance of the representations of similar things is **smaller than** the distance of dissimilar things

Similar like before, but apply on a triplet of [anchor (query), positive, negative] sample

Compare triplets in one go / learn to rank (order) samples

q

p

n

$$\|f(q) - f(p)\|^2 + m \leq \|f(q) - f(n)\|^2$$

$$d(q,p) + m \leq d(q,n)$$

$$d(q,p) + m - d(q,n) \leq 0$$

Ask for the negatives to be at least m further than the positives

Still Not A Learning algorithm

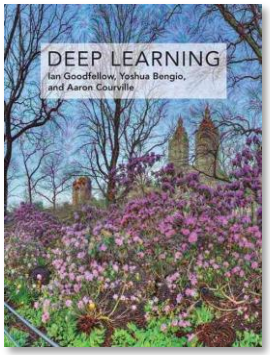
- We still need to understand
  - **Loss functions**: How to measure our error? This depends on the task we want to solve.
  - **Activation functions**: What kind of neurons should we use?
  - **Architectures**: How to combine neurons together to build meaningful models?
  - **Optimisation**: Is batch gradient descent the best way to use these error derivatives to discover a good set of weights?
  - **Regularisation**: How do we make sure we do not overfit?
  - **Initialisation**: Where do we start our search?

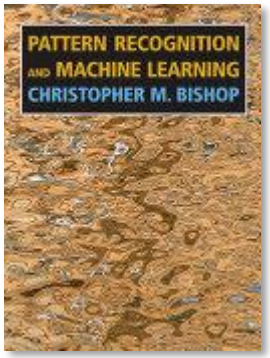
Define well-behaved landscapes

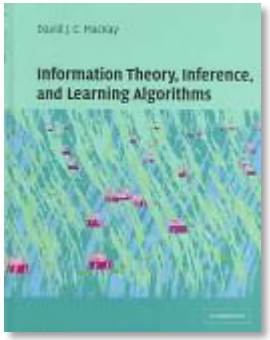
Efficient search

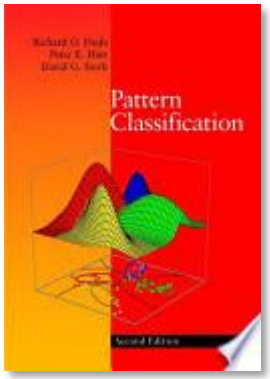
Resources (I)

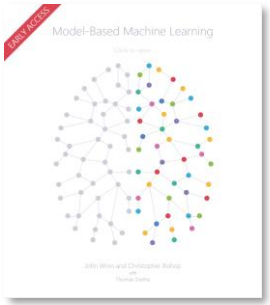
- *I. Goodfellow, Y. Bengio, A. Courville, "Deep Learning", MIT Press, 2016*
  http://www.deeplearningbook.org/
- *C. Bishop, "Pattern Recognition and Machine Learning", Springer, 2006*
  http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm
- *D. MacKay, "Information Theory, Inference and Learning Algorithms", Cambridge University Press, 2003*
  http://www.inference.phy.cam.ac.uk/mackay/
- *R.O. Duda, P.E. Hart, D.G. Stork, "Pattern Classification", Wiley & Sons, 2000*
  http://books.google.com/books/about/Pattern_Classification.html?id=Br33IRC3PkQC
- *J. Winn, C. Bishop, "Model-Based Machine Learning", early access*
  http://mbmlbook.com/











Further Info

- Many of the slides of these lectures have been adapted from various highly recommended online lectures and courses:
  - Andrew Ng's *Machine Learning Course*, Coursera
    https://www.coursera.org/course/ml
  - Andrew Ng's *Deep Learning Specialization,* Coursera
    https://www.coursera.org/specializations/deep-learning
  - Victor Lavrenko's *Machine Learning* Course
    https://www.youtube.com/channel/UCs7alOMRnxhzfKAJ4JjZ7Wg
  - Fei Fei Li and Andrej Karpathy's *Convolutional Neural Networks for Visual Recognition*
    http://cs231n.stanford.edu/
  - Geoff Hinton's *Neural Networks for Machine Learning, (*ex Coursera)
    https://www.youtube.com/playlist?list=PLiPvV5TNogxKKwvKb1RKwkq2hm7ZvpHz0
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ
  - Michael Nielsen's *Neural Networks and Deep Learning*
    http://neuralnetworksanddeeplearning.com/