NEURAL NETWORKS OVERVIEW

What is Deep Learning?

Artificial Intelligence

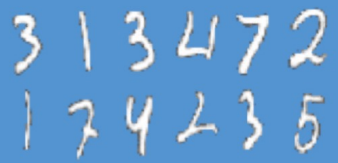
Any technique that enables computers to mimic human behavior

Machine Learning

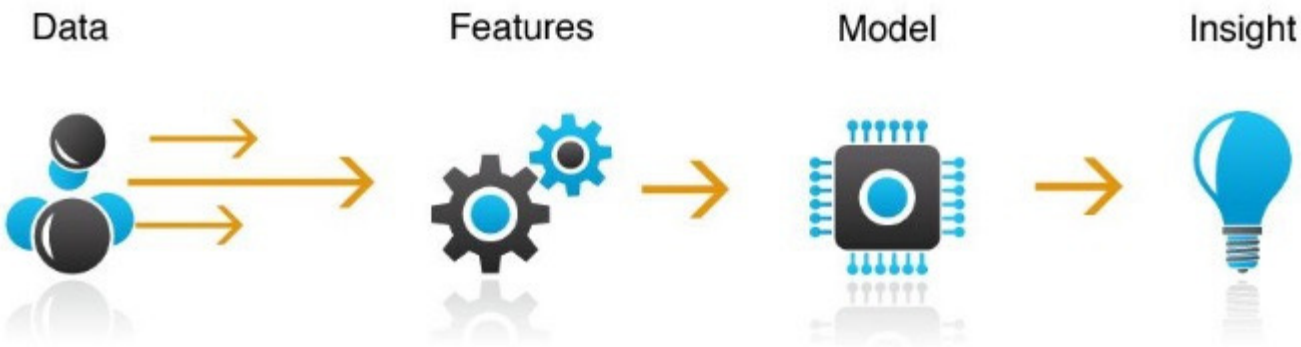
Ability to learn without explicitly being programmed

Deep Learning

Extract patterns from data using neural networks



SHALLOW LEARNING



Transforms the input data into **only one** or **two** successive **representation layers**

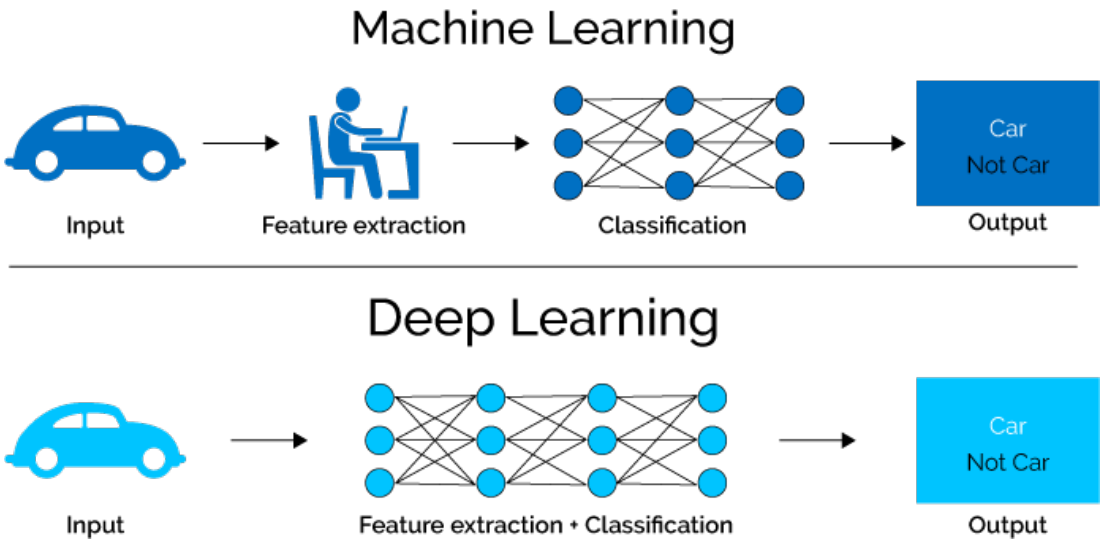
e.g. SVM: high-dimensional non-linear projections

But, complex problems require more refined representations

Feature Engineering = humans manually engineer layers of representations of the data

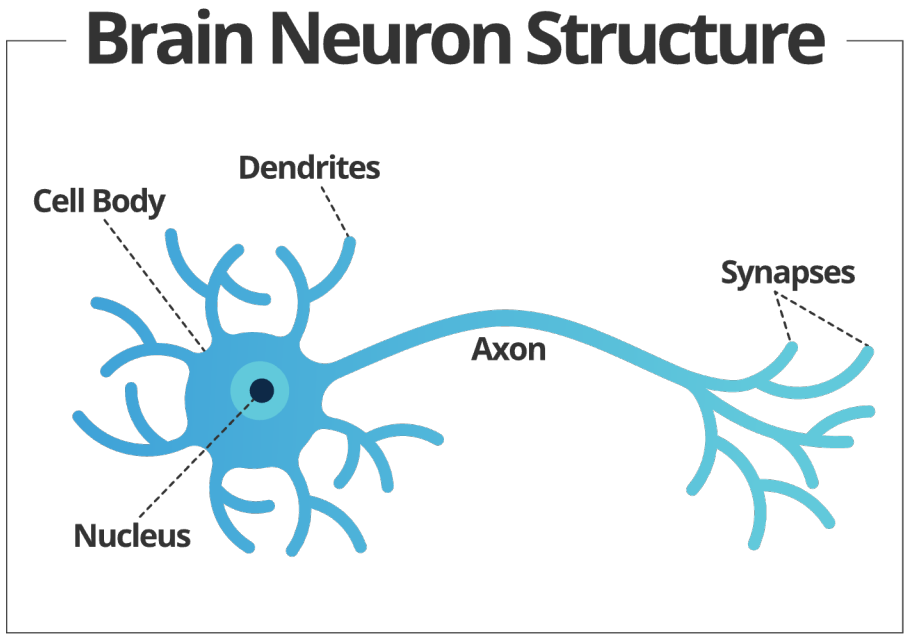
Good features -> solve the problem more elegantly, using fewer resources, less data

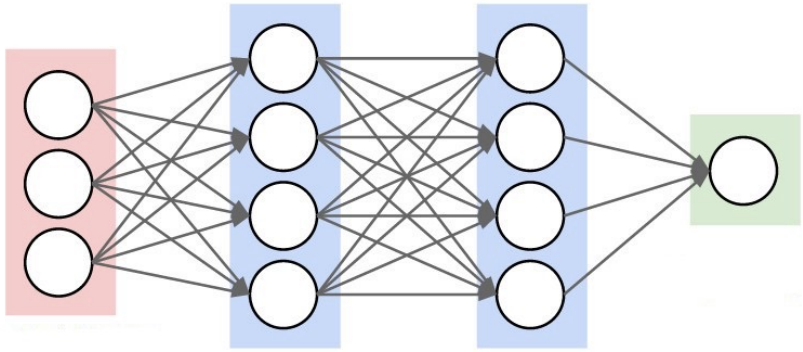
DEEP LEARNING



Deep Learning often involves **tens** or **hundreds** of successive **layers** of representations.
Hierarchical representation learning - increasingly complex representations are developed.
These intermediate incremental representations are **learned jointly**.
Deep Learning completely automates feature engineering

Artificial Neural Networks





Neural Networks

Good news:

The **universal approximation theorem** states that any continuous function $f : [0, 1]^n \rightarrow [0, 1]$ can be approximated arbitrarily well by a neural network with at least 1 hidden layer with a finite number of weights.

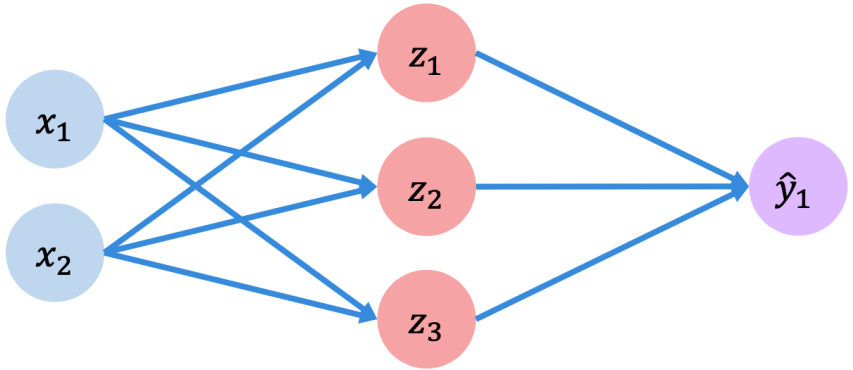
Bad news:

A feedforward network with a single layer is sufficient to represent any function, but **the layer may be infeasibly large** and **may fail to learn and generalize** correctly.

Common Loss Functions

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

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

Final Grades (percentage)

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

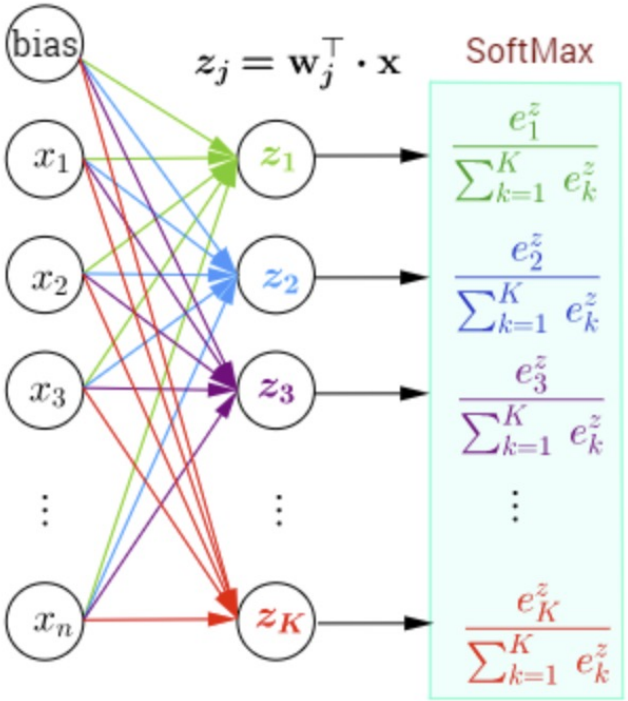
Actual Predicted

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

Common Loss Functions

$$\Phi(\overline{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{d} \exp(v_j)} \quad \forall i \in \{1 \dots d\}$$

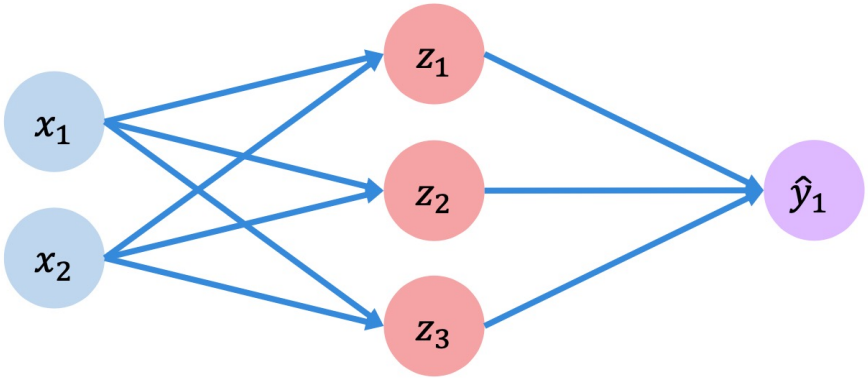
$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



Common Loss Functions

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

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$$

y

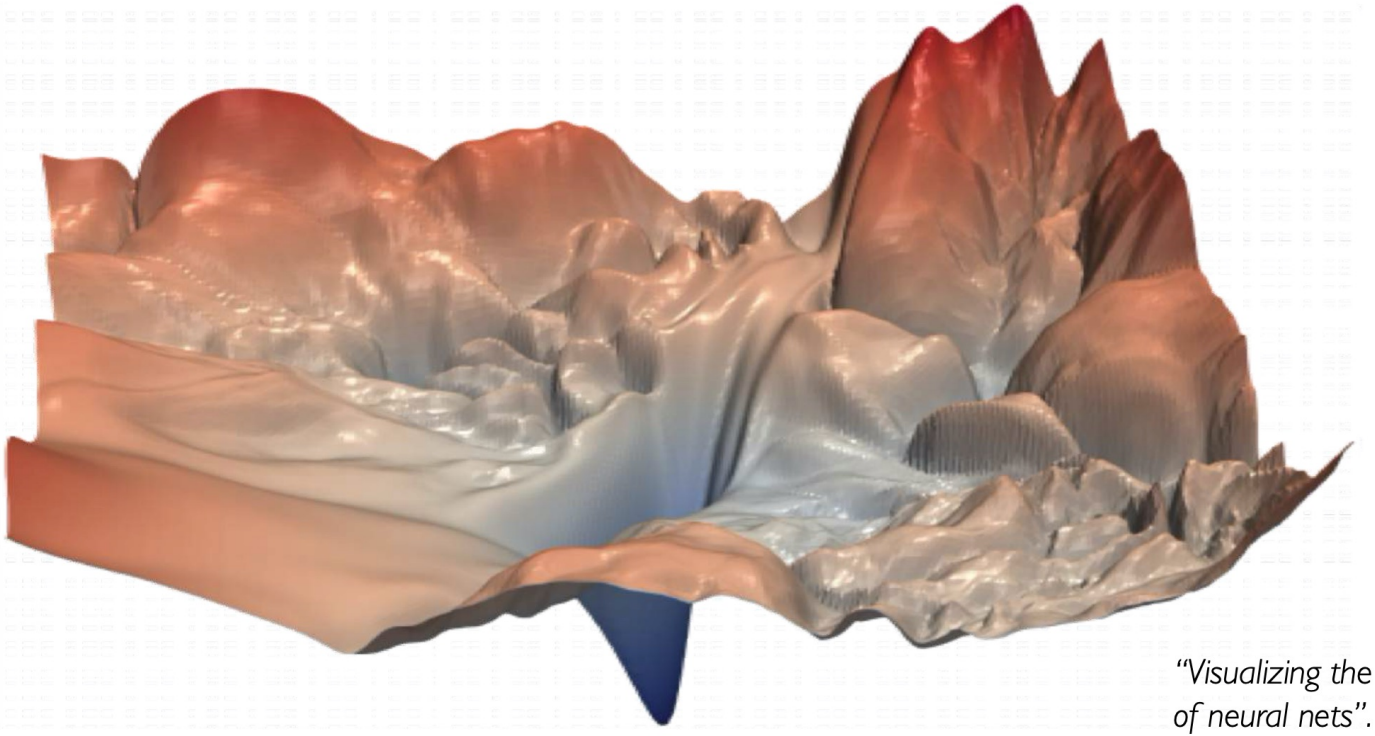
$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

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

Actual Predicted Actual Predicted

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

Loss Functions



"Visualizing the loss landscape of neural nets". Dec 2017.

Training Neural Networks with Gradient Descent

Algorithm

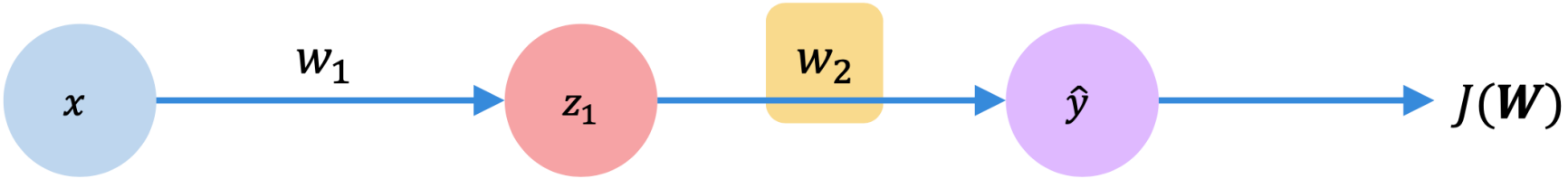
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

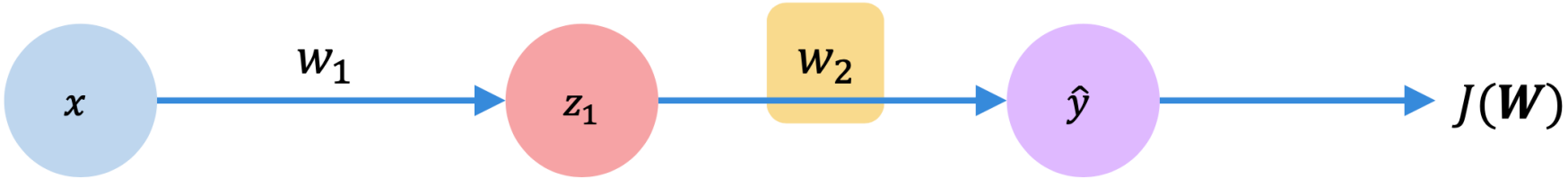
5. Return weights

Training Neural Networks: Backpropagation



How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?

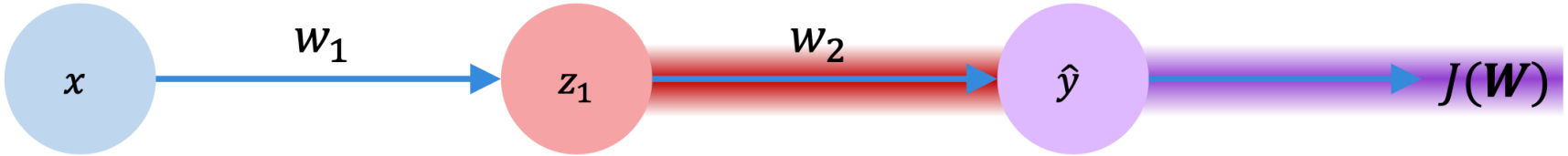
Training Neural Networks: Backpropagation



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

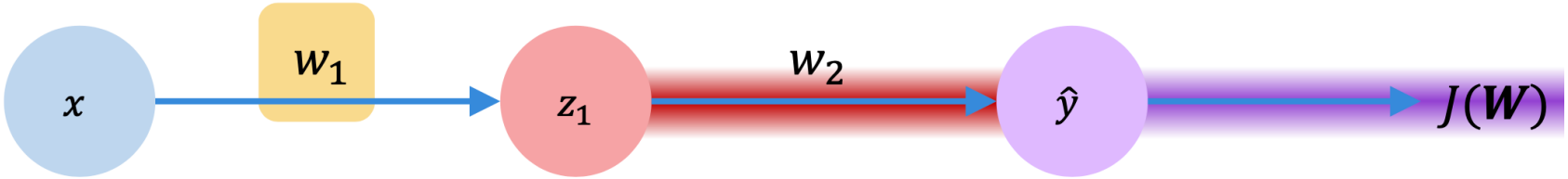
Let's use the chain rule!

Training Neural Networks: Backpropagation



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

Training Neural Networks: Backpropagation

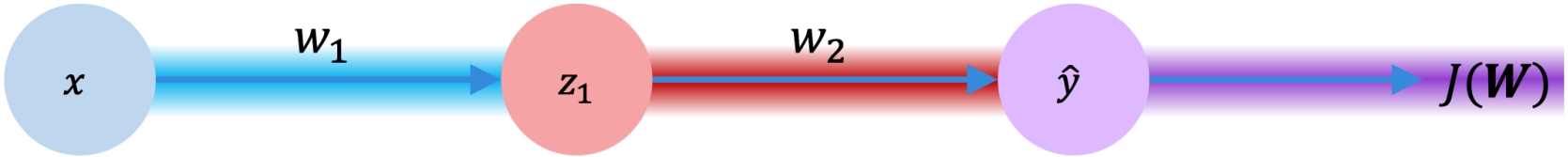


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

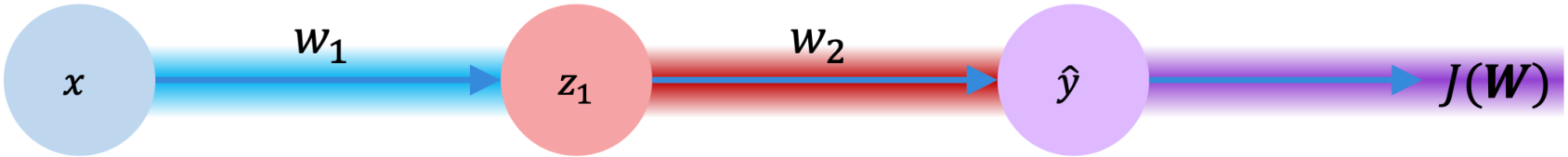
Apply chain rule!

Training Neural Networks: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

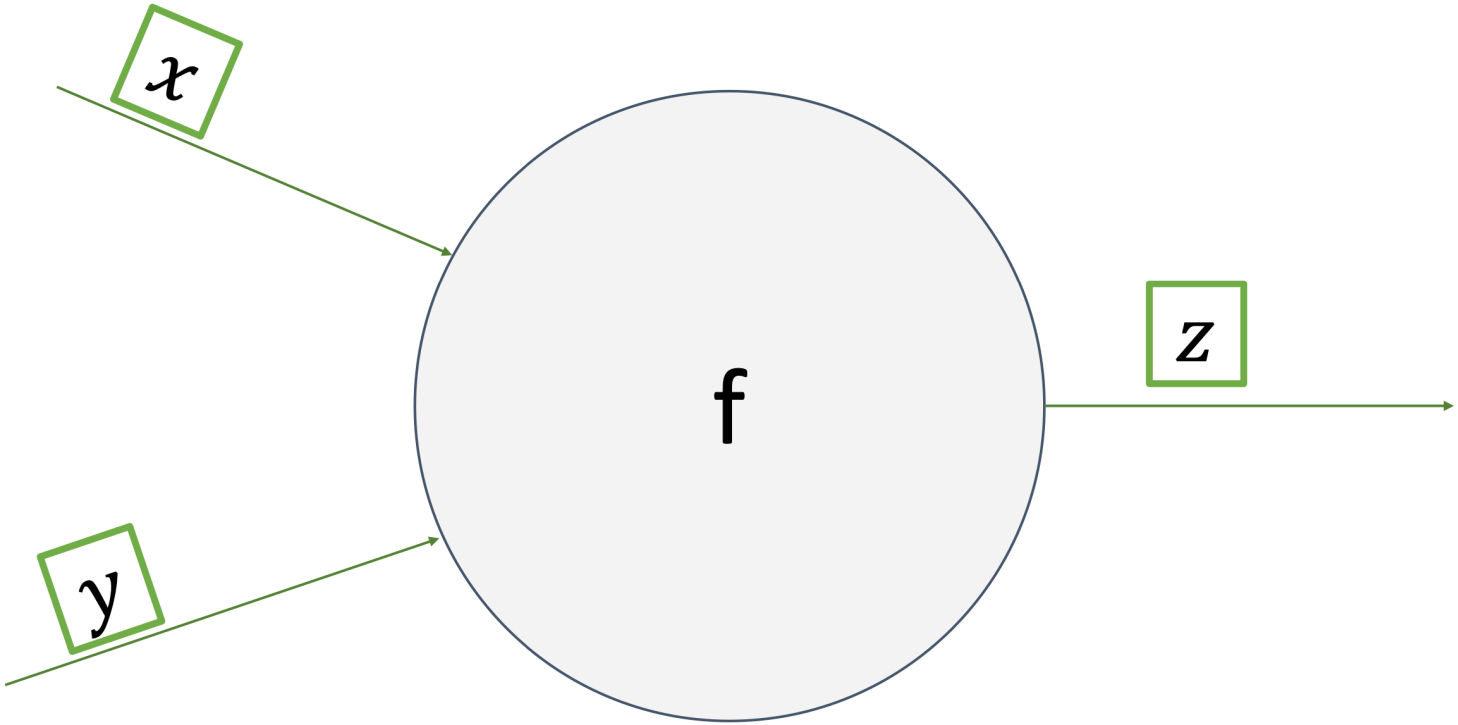
Training Neural Networks: Backpropagation



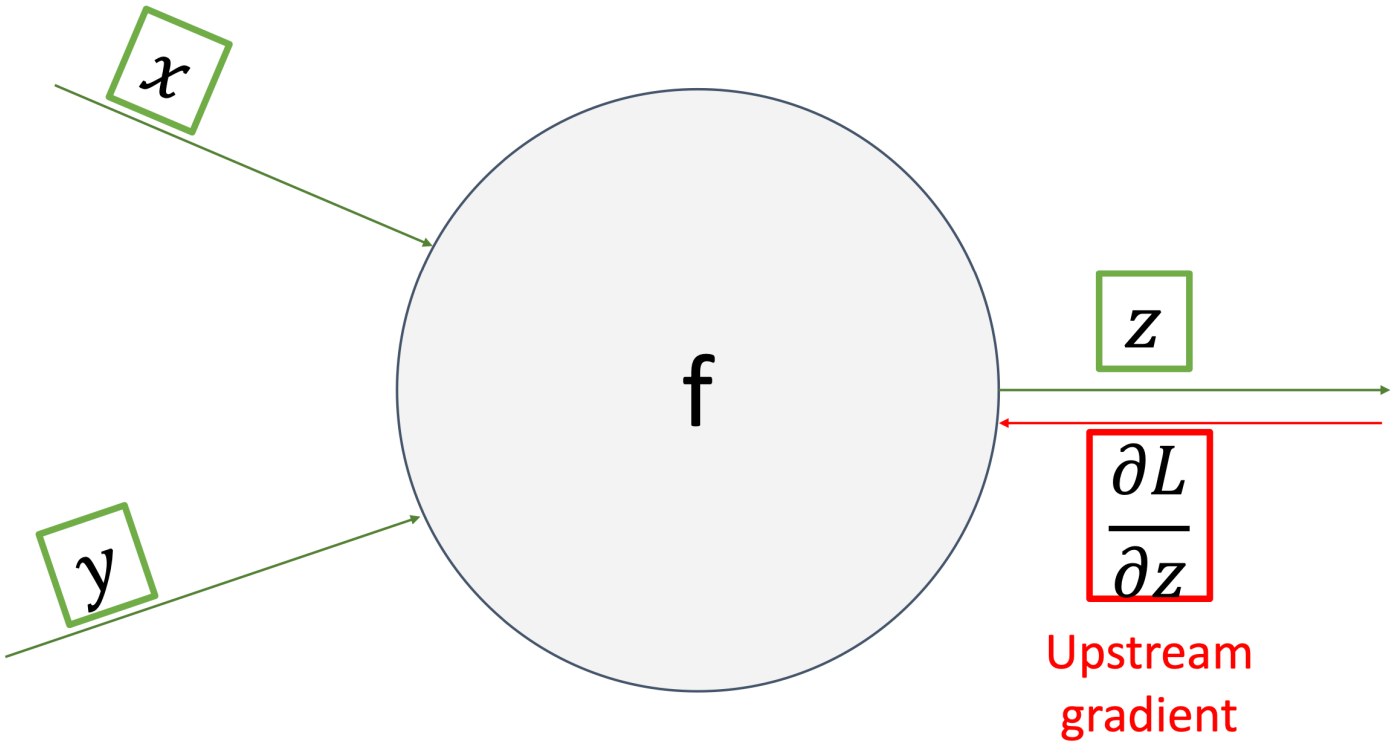
$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Repeat this for ***every weight in the network*** *using gradients from later layers*

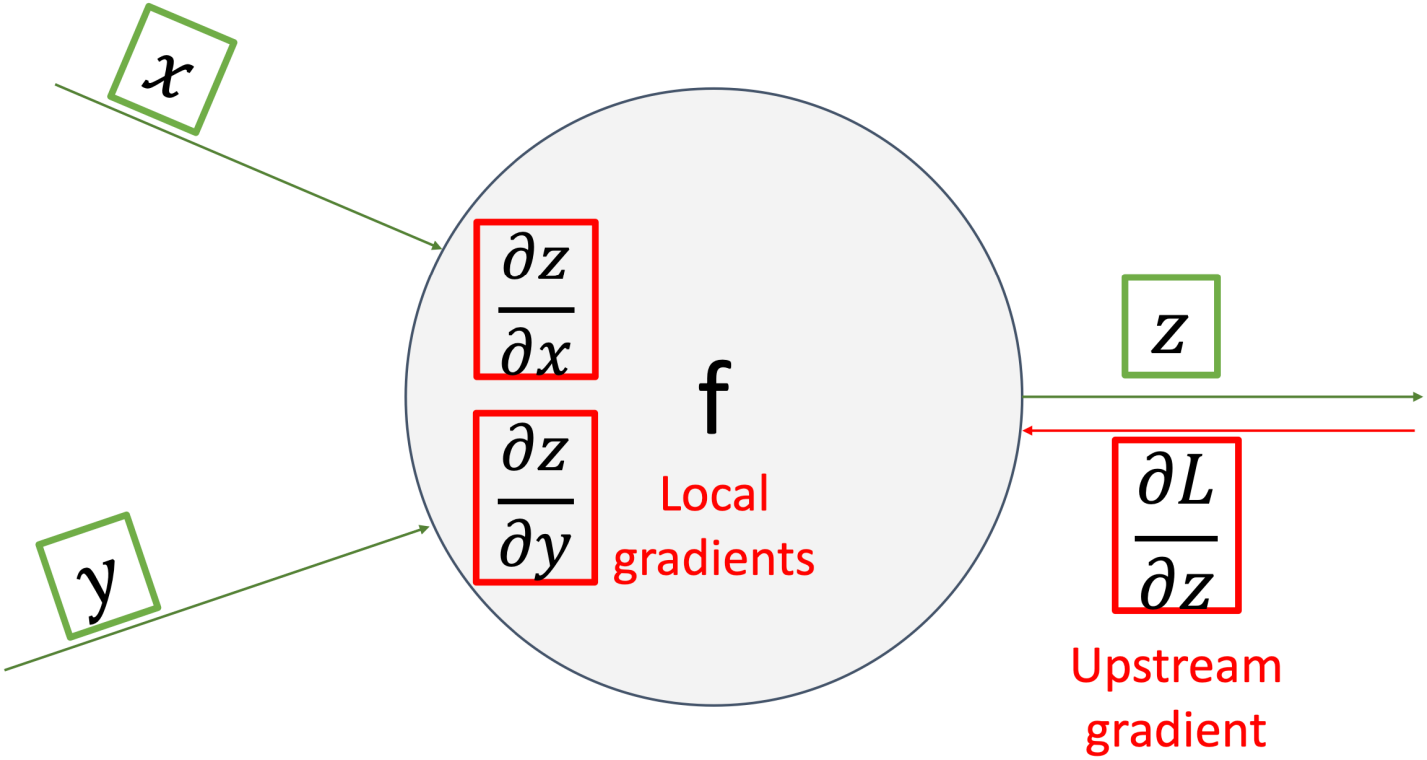
Backpropagation: Upstream Gradient



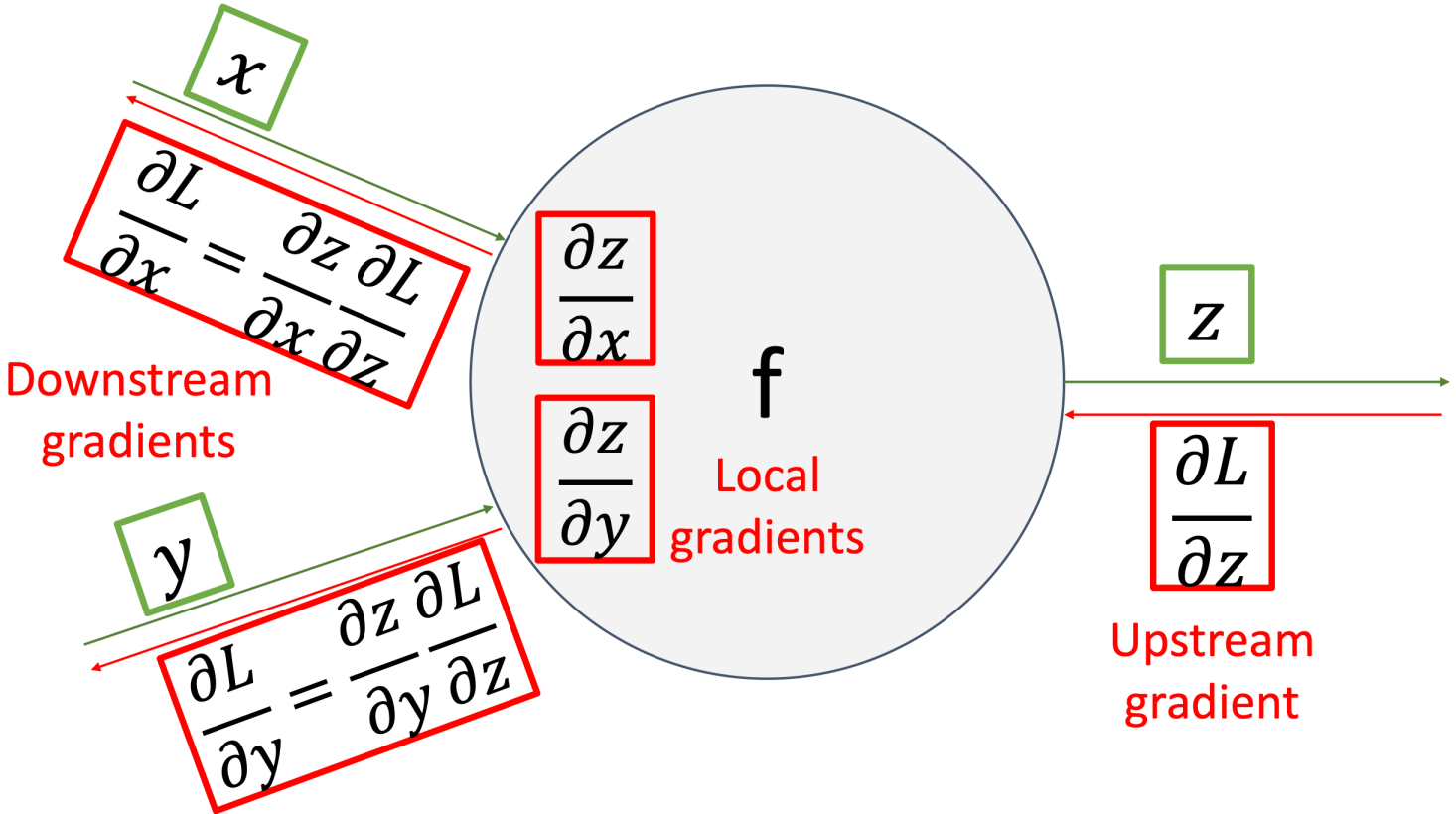
Backpropagation: Upstream Gradient



Backpropagation: Downstream/Upstream Gradient



Backpropagation: Upstream Gradient



Training Neural Networks with Adaptive Learning Rates

Learning rates need not be necessarily fixed

Can be made larger or smaller depending on:

- how large gradient is
- how fast learning is happening
- size of particular weights
- etc...

Training Neural Networks with Adaptive Learning Rates

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp

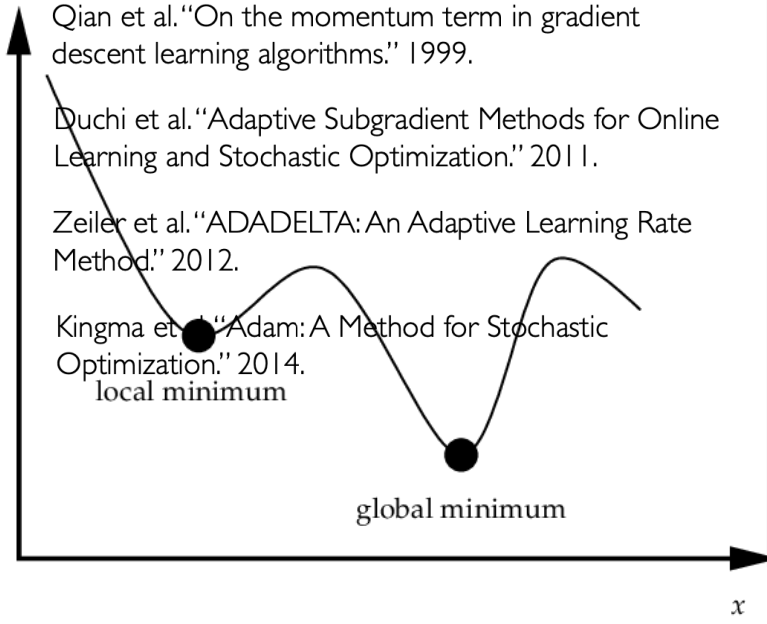
`tf.train.MomentumOptimizer`

`tf.train.AdagradOptimizer`

`tf.train.AdadeltaOptimizer`

`tf.train.AdamOptimizer`

`tf.train.RMSPropOptimizer`



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Training Neural Networks with 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 computational to compute!

Training Neural Networks with Mini-batch/Stochastic Gradient

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

Fast to compute and a much better estimate of the true gradient!

- Mini-batches lead to faster training
- Can be paralelized for GPUs

Sample, Batch, Epoch

- A **sample** is a single row of data. A sample may also be called an **instance**, an **observation**, an **input vector**, or a **feature vector**.

- The **batch size** is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. A training dataset can be divided into one or more batches.

- When *all training samples are used to create one batch*, the learning algorithm is called **batch gradient descent**. When the *batch is the size of one sample*, the learning algorithm is called (true) **stochastic gradient descent**. When the *batch size is more than one sample and less than the size of the training dataset*, the learning algorithm is called **mini-batch gradient descent**. In the case of mini-batch gradient descent, popular batch sizes include 32, 64, 128 and 256 samples.

- The number of **epochs** is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.

Some recent developments..

Gradients without Backpropagation

Atılım Güneş Baydin [1] Barak A. Pearlmutter [2] Don Syme [3] Frank Wood [4] Philip Torr [5]

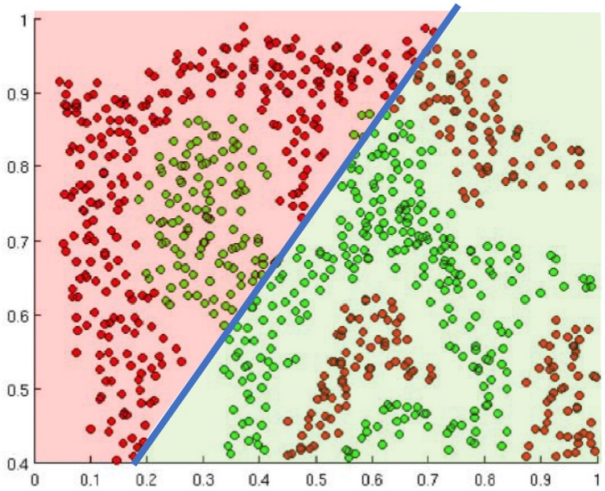
Abstract

Using backpropagation to compute gradients of objective functions for optimization has remained a mainstay of machine learning. Backpropagation, or reverse-mode differentiation, is a special case within the general family of automatic differentiation algorithms that also includes the forward mode. We present a method to compute gradients based solely on the directional derivative that one can compute exactly and efficiently via the forward mode. We call this formulation the **forward gradient**, an unbiased estimate of the gradient that can be evaluated in a single forward run of the function, entirely eliminating the need for backpropagation in gradient descent. We demonstrate forward gradient descent in a range of problems, showing substantial savings in computation and enabling training up to twice as fast in some cases.

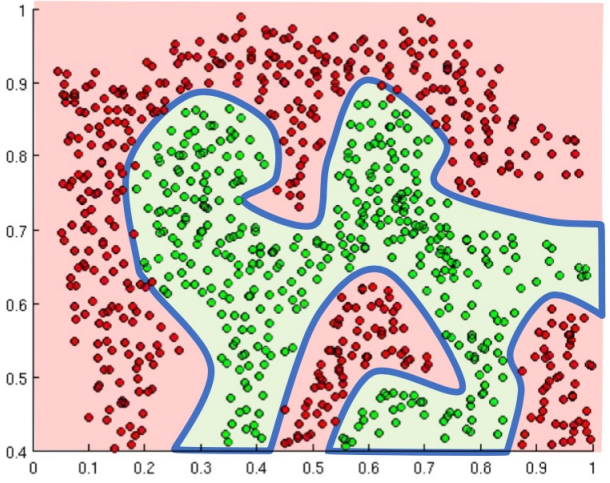
forward–backward algorithm, of which backpropagation is a special case conventionally applied to neural networks. This is mainly due to the central role of scalar-valued objectives in ML, whose gradient with respect to a very large number of inputs can be evaluated exactly and efficiently with a single evaluation of the reverse mode.

Reverse mode is a member of a larger family of AD algorithms that also includes the forward mode (Wengert, 1964), which has the favorable characteristic of requiring only a single forward evaluation of a function (i.e., not involving any backpropagation) at a significantly lower computational cost. Crucially, forward and reverse modes of AD evaluate different quantities. Given a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$, forward mode evaluates the Jacobian–vector product $\boldsymbol{J_f v}$, where $\boldsymbol{J_f} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{v} \in \mathbb{R}^n$; and revese mode evaluates the vector–Jacobian product $\boldsymbol{v}^\intercal \boldsymbol{J_f}$, where $\boldsymbol{v} \in \mathbb{R}^m$. For the case of $f : \mathbb{R}^n \to \mathbb{R}$ (e.g., an objective function in ML), forward mode gives us $\nabla f \cdot \boldsymbol{v} \in \mathbb{R}$. the directional derivative:

Activation Functions



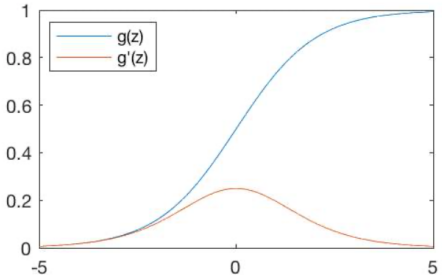
Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

Common Activation Functions

Sigmoid Function

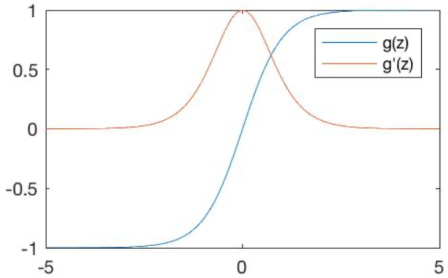


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

`tf.nn.sigmoid(z)`

Hyperbolic Tangent

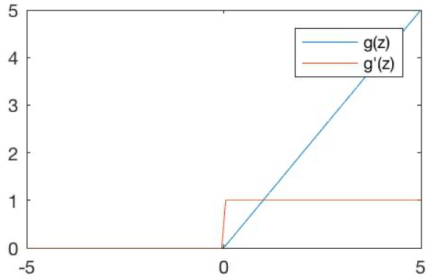


$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

`tf.nn.tanh(z)`

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

`tf.nn.relu(z)`

NOTE: All activation functions are non-linear

Epoch
000,000

Learning rate: 0.03

Activation: Tanh

Regularization: None

Regularization rate: 0

Problem type: Classification

http://playground.tensorflow.org/

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

Which properties do you want to feed in?

X_1

X_2

X_1^2

X_2^2

X_1X_2

$\sin(X_1)$

$\sin(X_2)$

2 HIDDEN LAYERS

4 neurons

2 neurons

*This is the output from one **neuron**. Hover to see it larger.*

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

OUTPUT

Test loss 0.495

Training loss 0.510

Colors shows data, neuron and weight values.

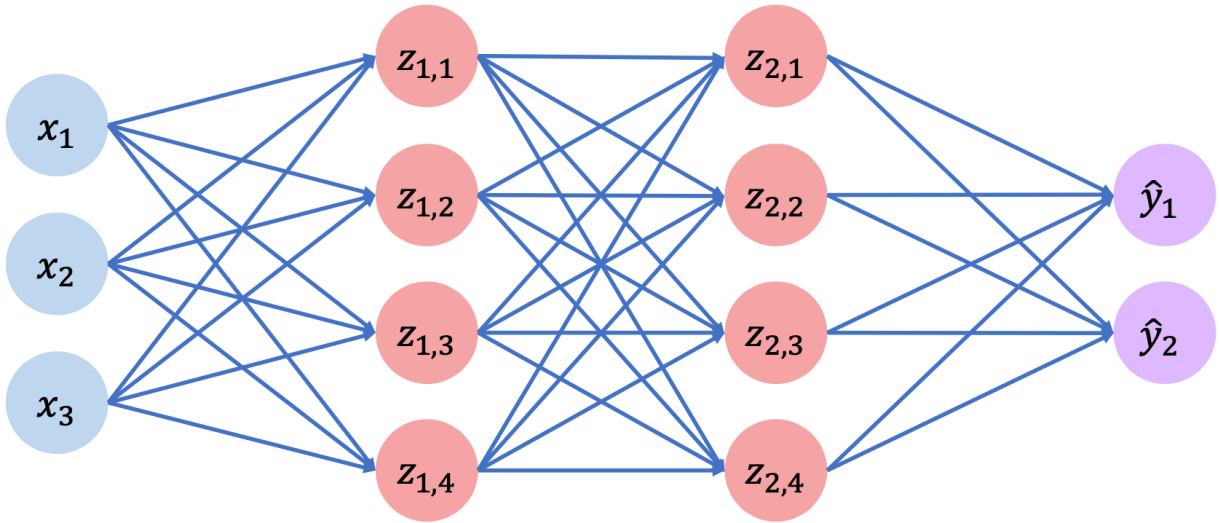
-1 0 1

Show test data

Discretize output

Regularization: Dropout

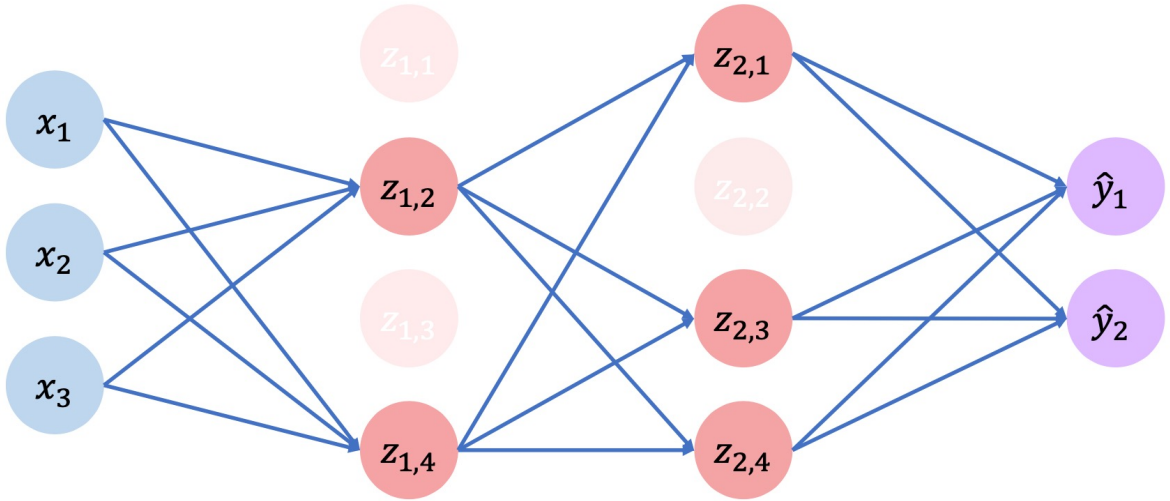
- During training, randomly set some activations to 0



Regularization: Dropout

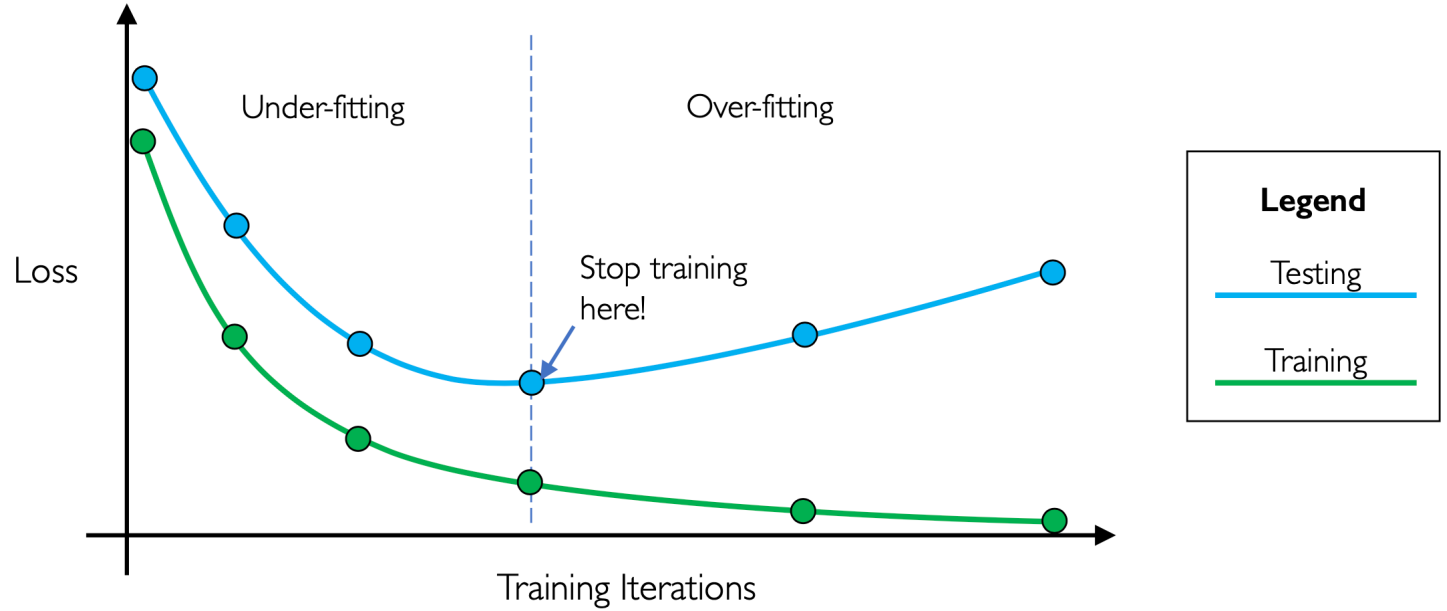
- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

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



Regularization: Early Stopping

- Stop training before we have a chance to overfit



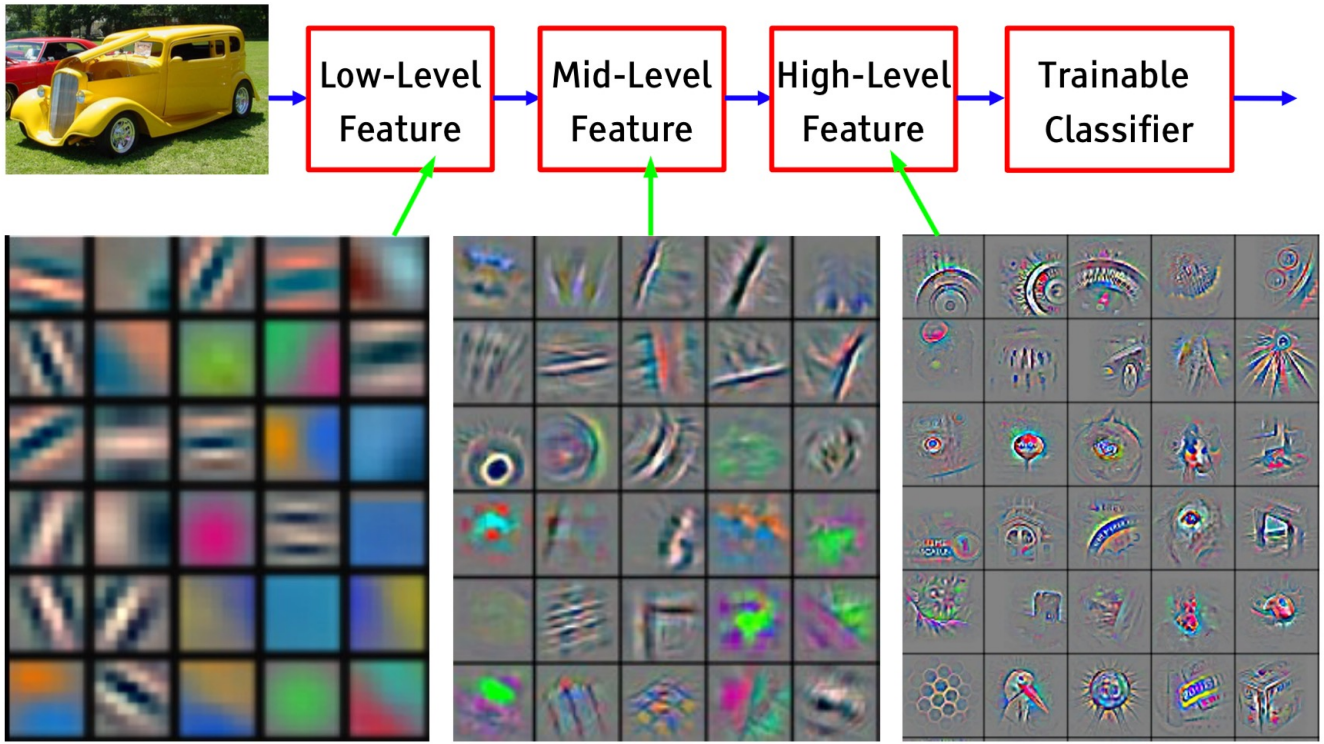
What training typically does not do

Choice of the hyper-parameters has to be done manually:

- Type of activation function
- Choice of architecture (how many hidden layers, their sizes)
- Learning rate, number of training epochs
- What features are presented at the input layer
- How to regularize

It may seem complicated at first - the best way to start is to re-use some existing setup and try your own modifications.

Convolutional Neural Networks (CNNs)



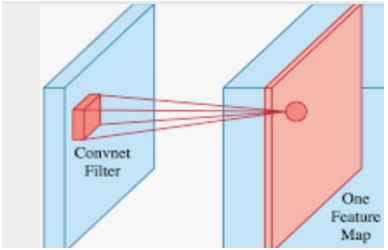
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Convolutional Neural Networks (CNNs)

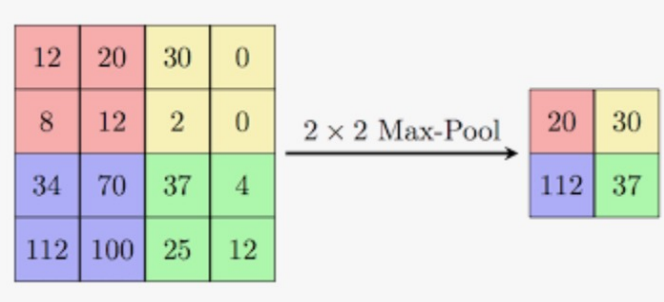
- Practical for computer vision applications
- Hidden layer neurons are not fully connected
- Reduce number of weights (better to avoid overfitting)
- Learn same features irrespective of location in image (edges, eyes, faces…) - concept of weight sharing
- Use same Learning rules (Backpropagation and gradient descent)

Main Layers in a CNN

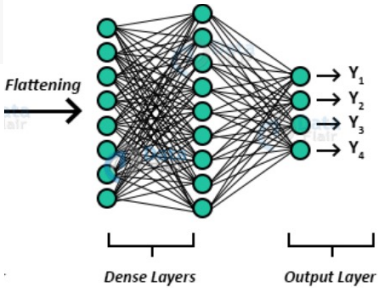
- Convolution Layer



- Pooling Layer



- Dense Layer (Fully-connected layer)



Could be used in classification tasks such as Sentiment Analysis, Spam Detection or Topic Categorization

Summary

Neural Networks reviewed:

- Loss functions
- Backpropagation
- Activation Functions
- Regularization
- CNNs

Thanks!
Questions...