Lecture 3
Neural networks

CS 180 – Intelligent Systems

Dr. Victor Chen

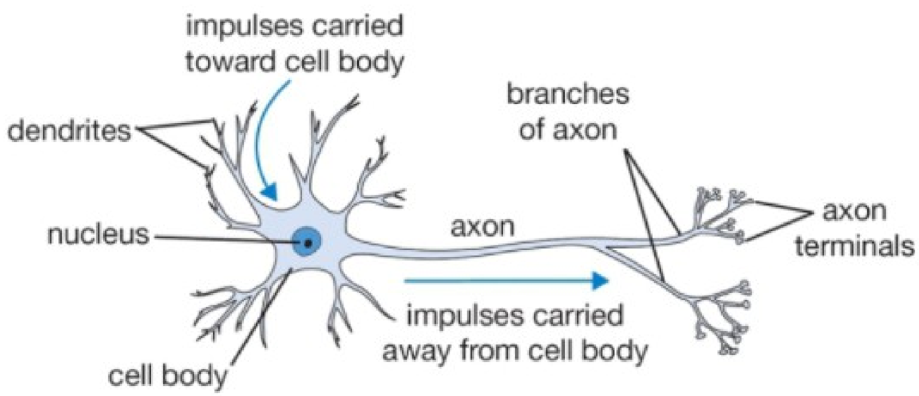
Spring 2021

Neural networks

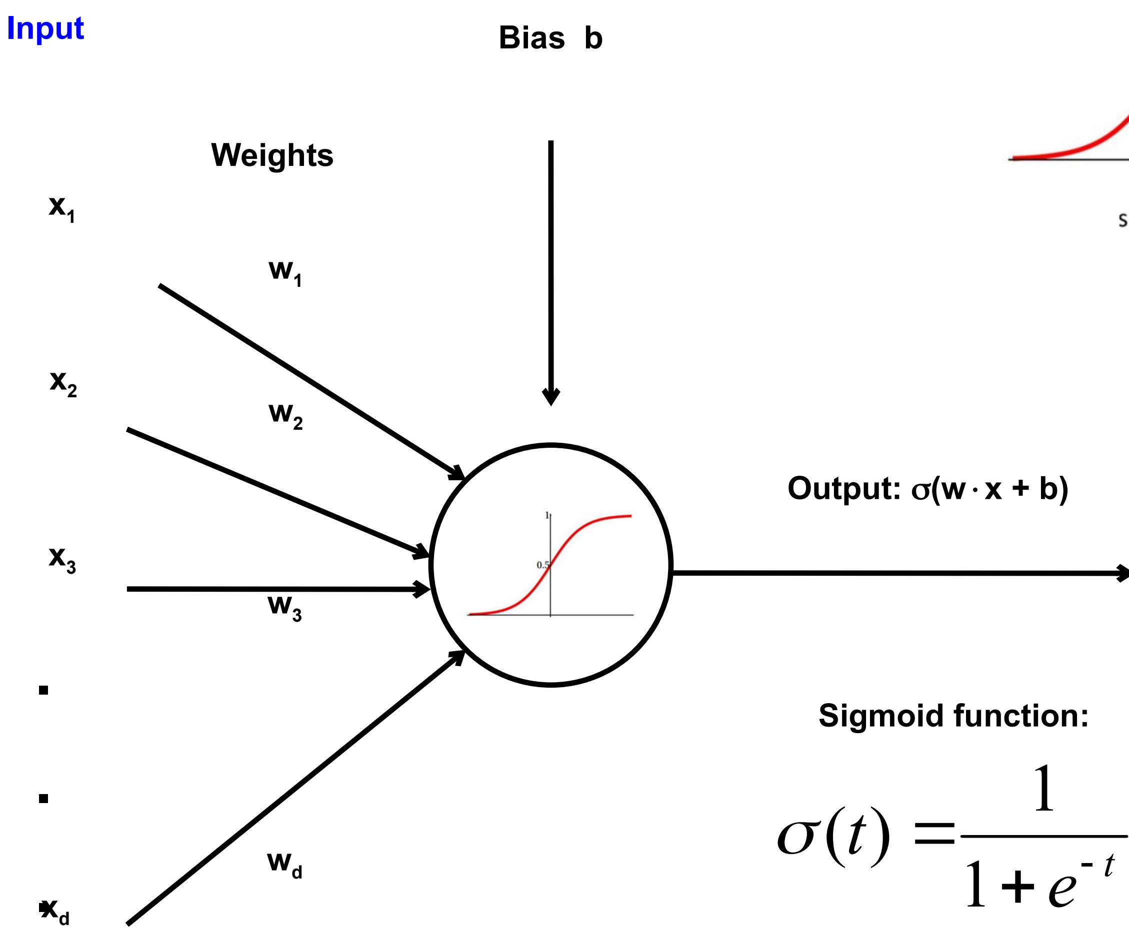
Neural networks

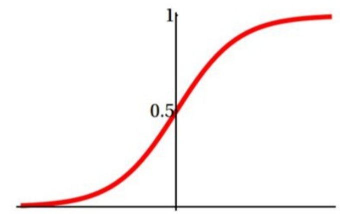
Neural networks are collections of neurons.

One neuron is a weighted sum, followed by an activation function.



Neuron with sigmoid = Logistic regression



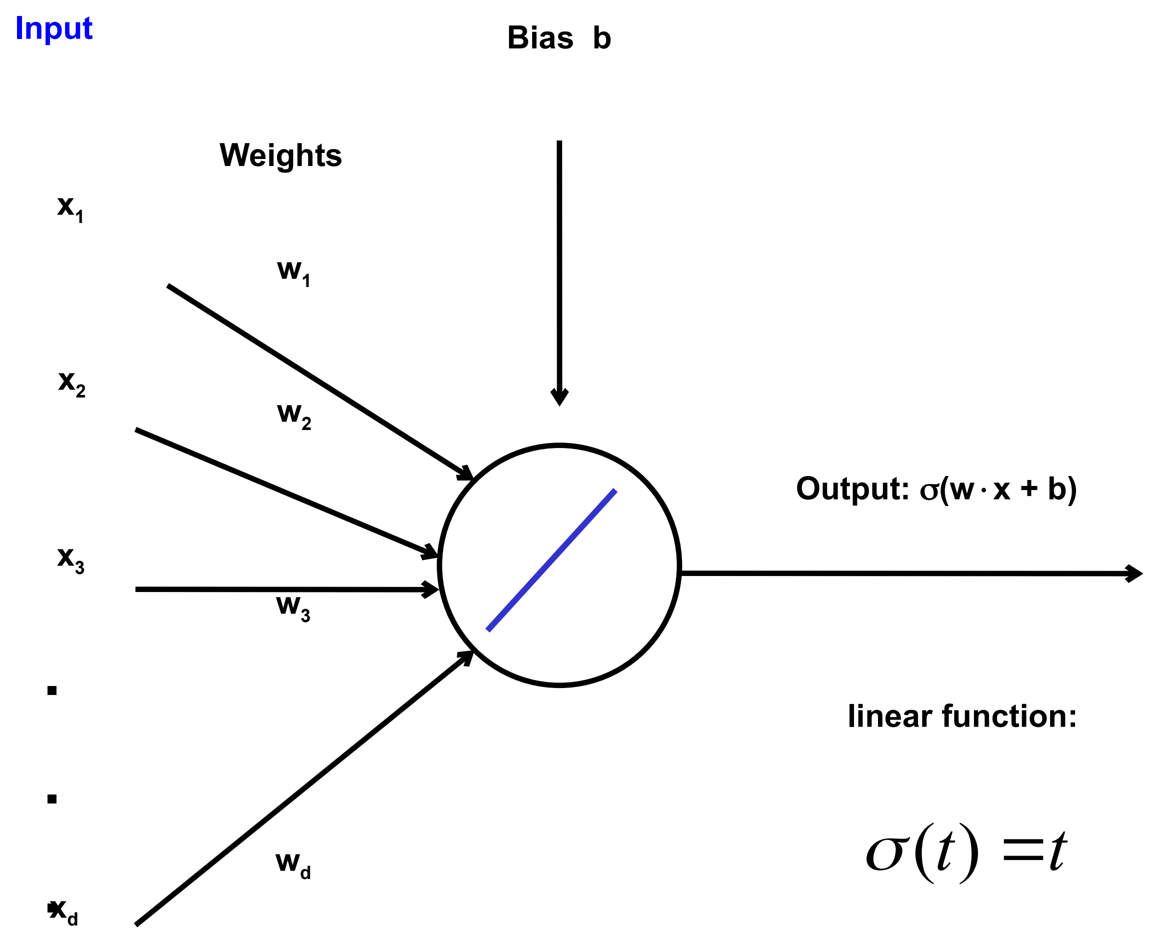


sigmoid

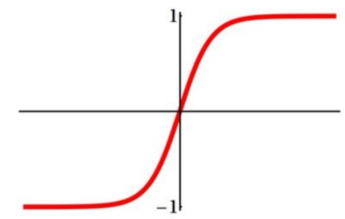
Sigmoid function:

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

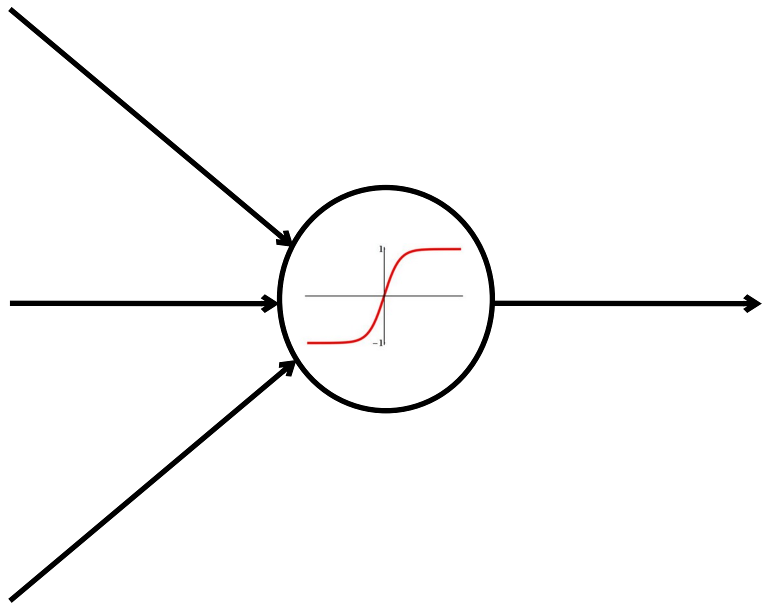
Neuron with linear function = Linear regression



Neuron with Tanh function

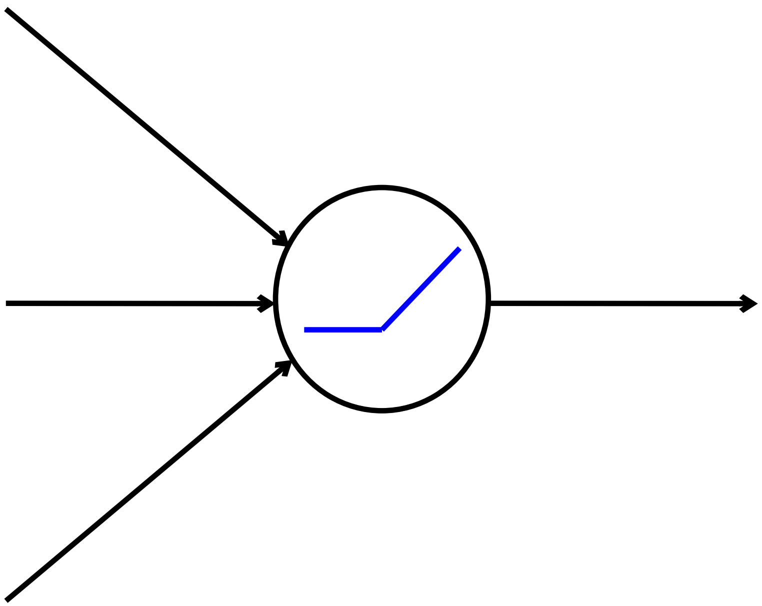


tanh



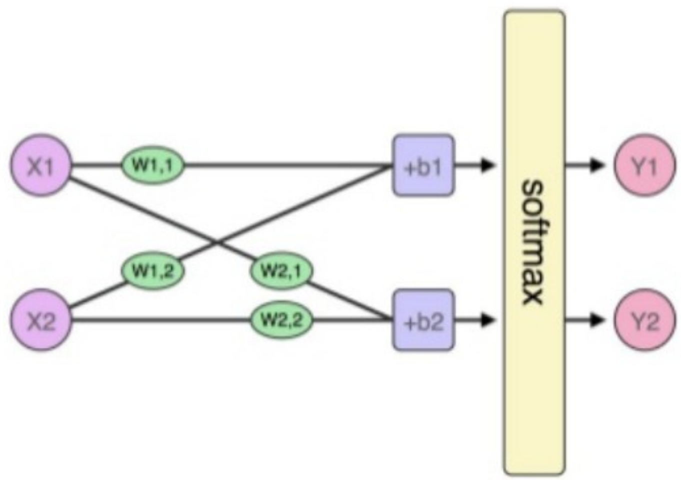
Neuron with ReLU function

Rectified linear unit (ReLU): $\sigma(t) = \max(0, t)$



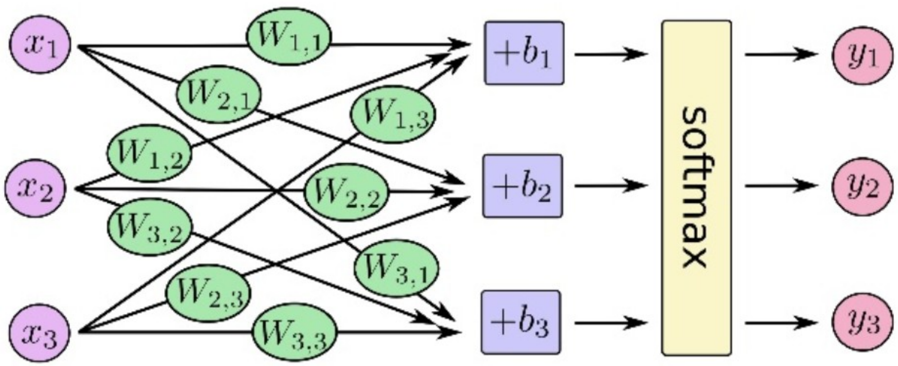
Model with two outputs

We can put two neurons in parallel



Model with three outputs

We can put three neurons in parallel



Softmax implementation

$$P(y=j \mid \Theta^{(i)}) = \frac{e^{\Theta^{(i)}}}{\sum_{j=0}^{k} e^{\Theta_k^{(i)}}}$$

Softmax function

```python
def softmax(x):
    """Compute the softmax of vector x."""
    exps = np.exp(x)
    return exps / np.sum(exps)
```

np.exp(x): Calculate the exponential of x

Softmax example

What is softmax([1, 2, 3])?

y1 =   = 0.09

y2 =   = 0.24

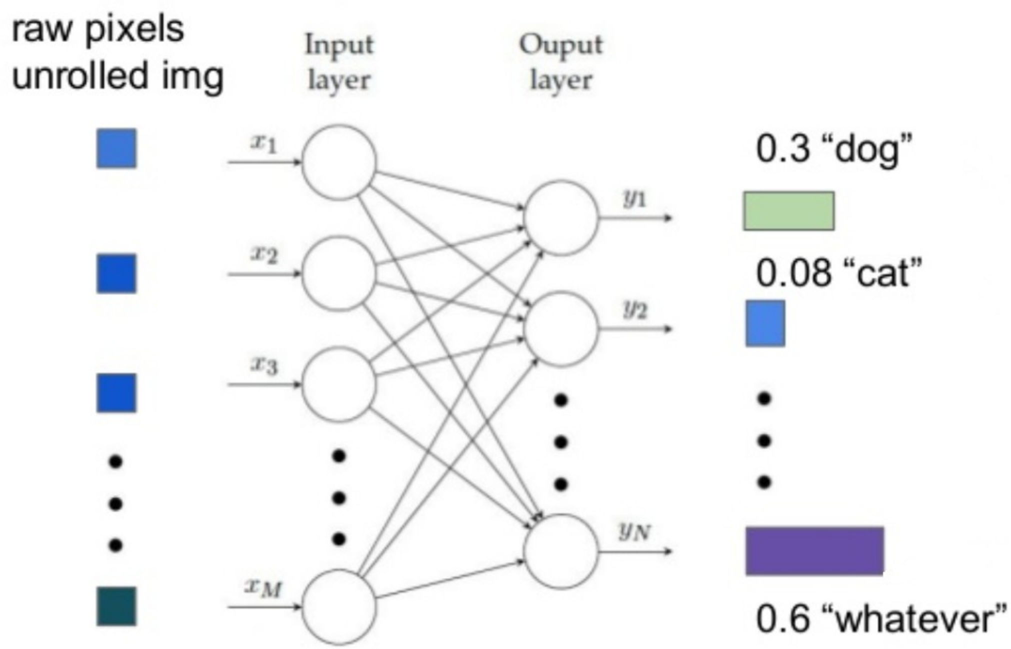
y3 = = 0.67

Output vector:

[0.09, 0.24, 0.67]

Neutral Network for image classification

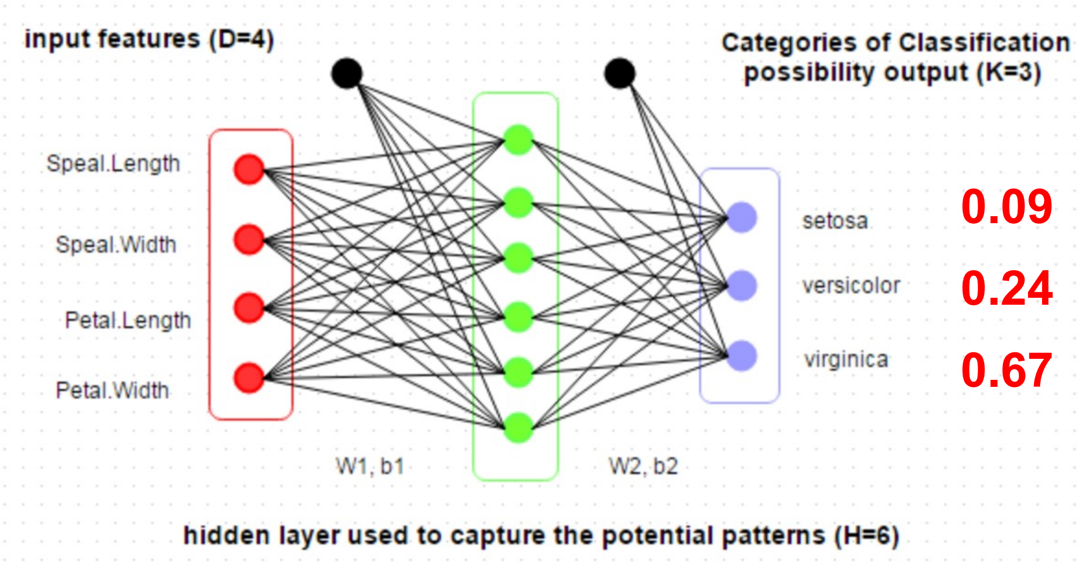
Build a multi-layer neural network! One output neuron for one class



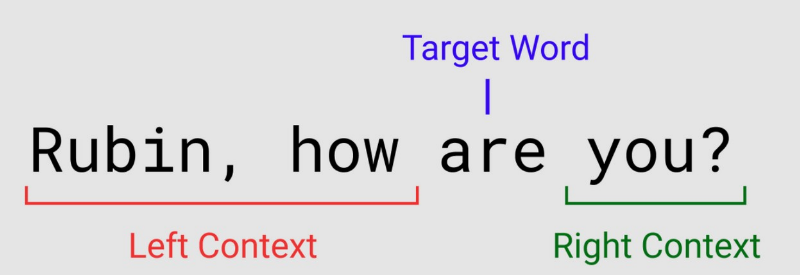
Softmax

Softmax function is used to ensure that the model output will be a probabilistic distribution, i.e., it converts any given vector to a distribution.

setosa, versicolor, virginica?



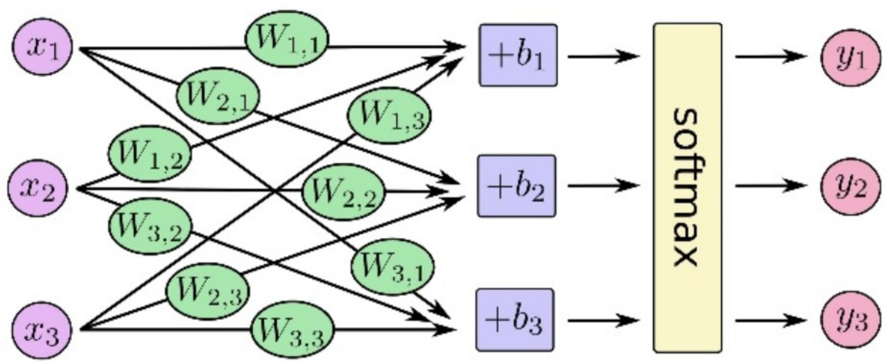
Neutral network for language processing



Training of multi-layer networks

- Goal: Find edge weights to minimize the error between "true" and "predicted" labels of training records:

$$E(\mathbf{w}) = \sum_{j=1}^{N} \left(y_j - f_{\mathbf{w}}(\mathbf{x}_j)\right)^2$$



Training of multi-layer networks

https://ml4a.github.io/demos/simple_forward_pass/

https://lecture-demo.ira.uka.de/neural-network-demo/

How to update model weights: gradient descent

- **Update weights by gradient descent:**

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \boxed{\frac{\partial E}{\partial \mathbf{w}}}$$

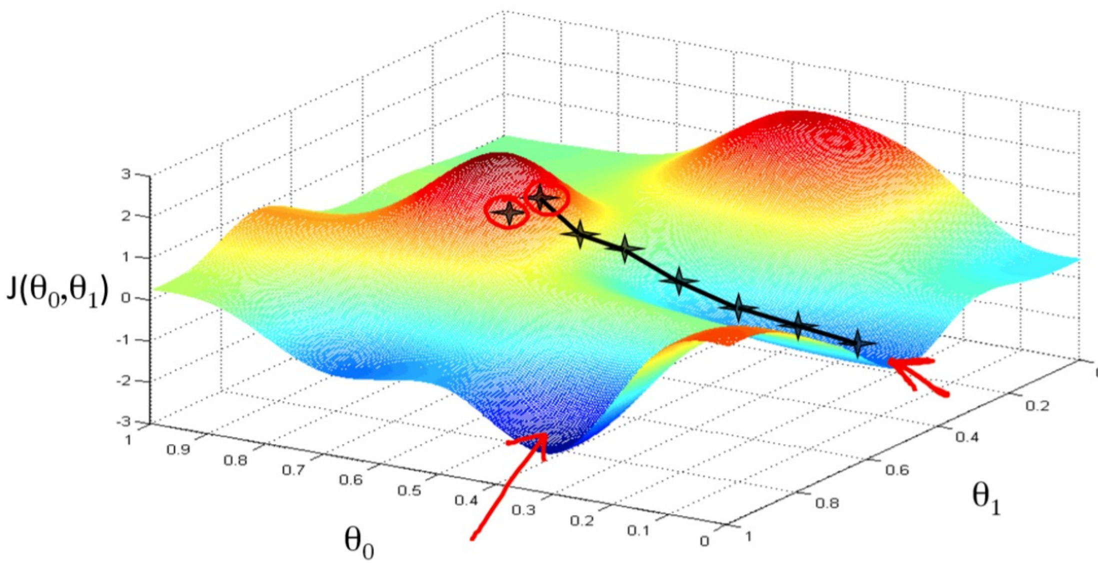
Gradient update

- **α here is called learning rate.**

Gradient descent

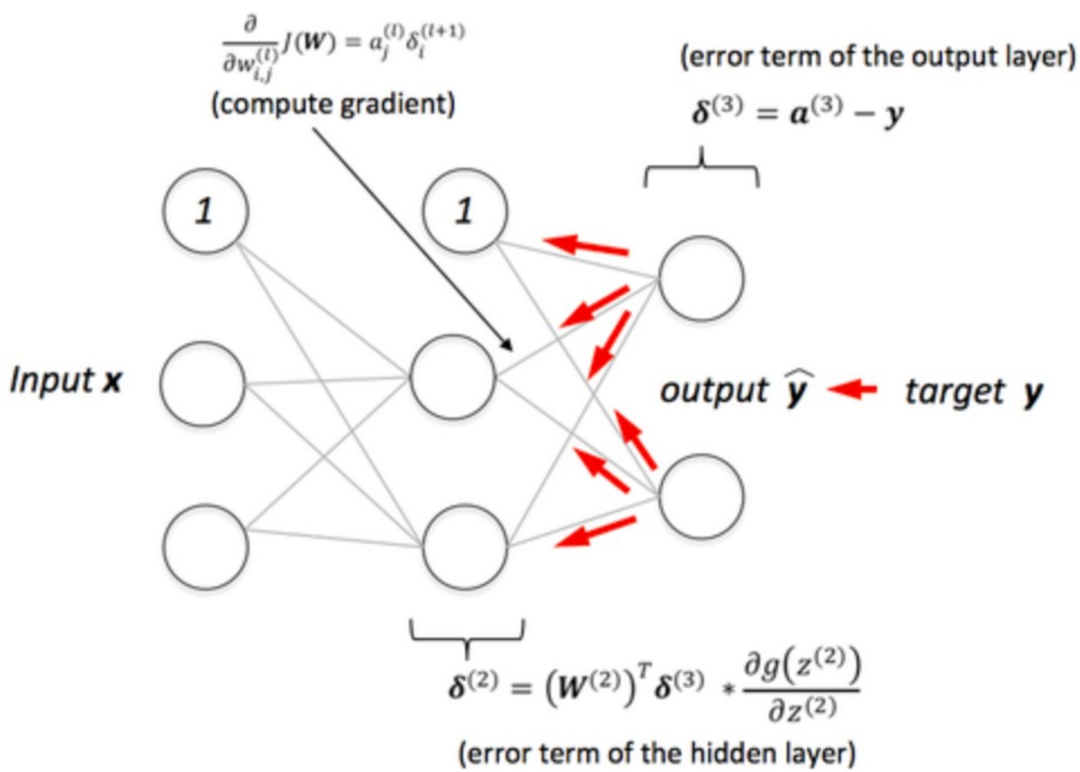
```
update = learning_rate * gradient_of_parameters

parameters = parameters - update
```



Back-propagation (gradient descent)

Gradient updates are computed in the direction from output to input layers.



Two key questions for back-propagation

1. How often the weights will be updated?

2. How the weights will be updated at each step?

How often weights are updated?

When you train a model, you need to specify how many samples you want to use per update. That is called **batch size.**

- **Online Training** - Update the weights based on gradients from a single training record.
- **Vanilla Gradient Descent** - Update the weights based on the gradients over all training records.
- **Batch Size Training**- Update the weights at a time based on some batch size of training elements.
- **Mini-Batch Training** - The same as batch size, but with a very small batch size. (Batch size defaults to 32 in Tensorflow/Keras).

Step v.s. Epoch

- The batch size is smaller than the training data size, so it may take several batches to make it completely through the training set.

- At **each step (iteration),** one batch was processed.

- At **each epoch**, we went through the complete training set once. A single pass through the entire training set

How weights are updated per step?

Choose **update rules (optimizers).** There are a few variations of gradient descent you can choose from

TensorFlow allows the update rule to be set to one of:

- **Adam**
- **Adagrad**
- **Adadelta**
- **Adamax**
- **Momentum**
- **Stochastic Gradient Descent (SGD)**
- Others

https://keras.io/optimizers/

Reflection on update rules (adaptive vs non-adaptive)

Each algorithm has its strengths and weaknesses.

- For rapid prototyping, use adaptive techniques like Adam/Adagrad. These help in getting quicker results with much less efforts.

- To get the best results, you should use Stochastic Gradient Descent (SGD). SGD is slow to get the desired results, but these results are mostly better than adaptive techniques.

Neural networks: pros and cons

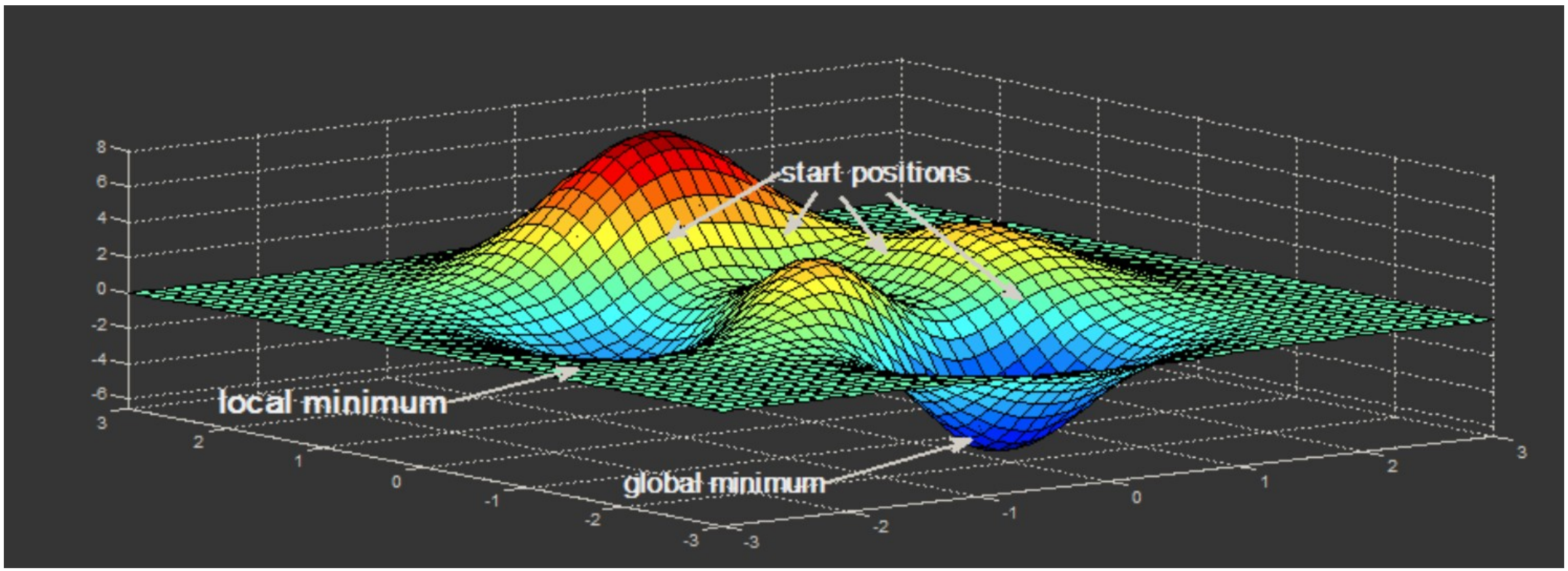
Pros

- Flexible and general function approximation framework
- Can build extremely powerful models by adding more layers

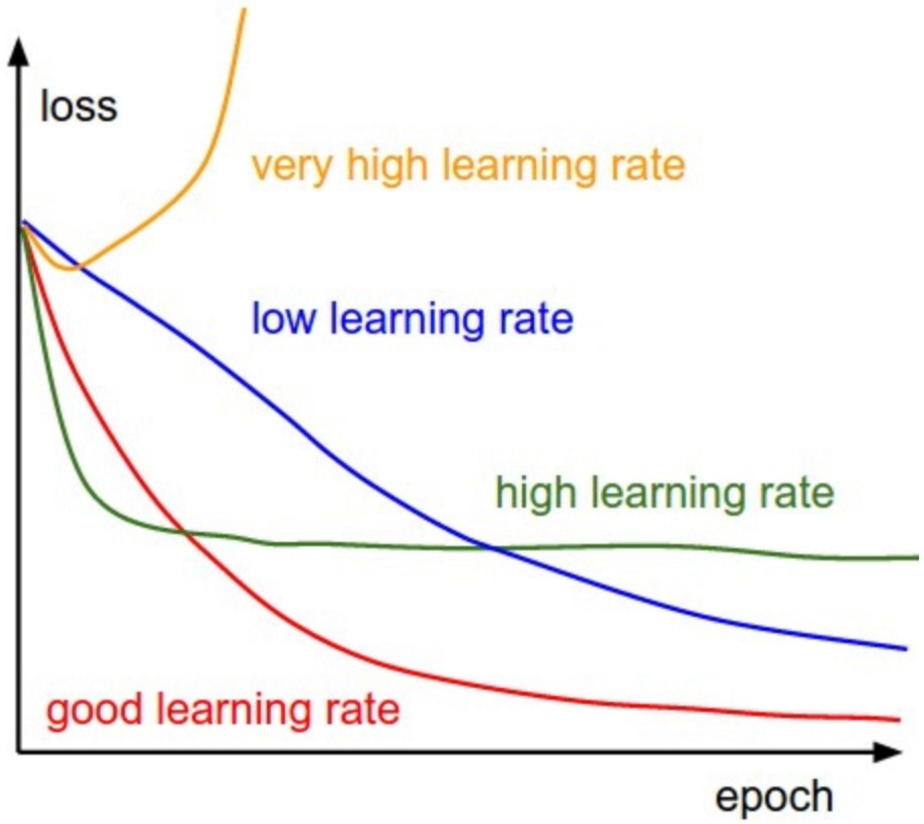
Cons

- Training is prone to local optimum.
- Huge amount of training data, computing power required to get good performance
- Implementation choices is huge (network architectures, parameters)

Gradient descent is prone to local optimum



Pay attention to learning rate



https://www.benfrederickson.com/numerical-optimization/

Demo of gradient descent

https://lukaszkujawa.github.io/gradient-descent.html

Neutral Network for Image Classification

https://portal.valossa.com/portal/image-recognition

Neutral network for image captioning

http://noteworthy.liacs.nl/