



Feedforward neural networks

- The building blocks of deep learning

MMAI 5500 – lecture 2

Fall 2024



Contents

The forward pass

Activation functions

Loss functions

The backward pass

- Gradients
- The chain rule

Weights update/optimization

- Stochastic gradient descent
- Learning rate
- Momentum

Two fundamentally different neural network architectures

Neural networks

- two different architectures

Feed-forward networks

Fixed size inputs

Popular for computer vision

E.g. CNN or MLP

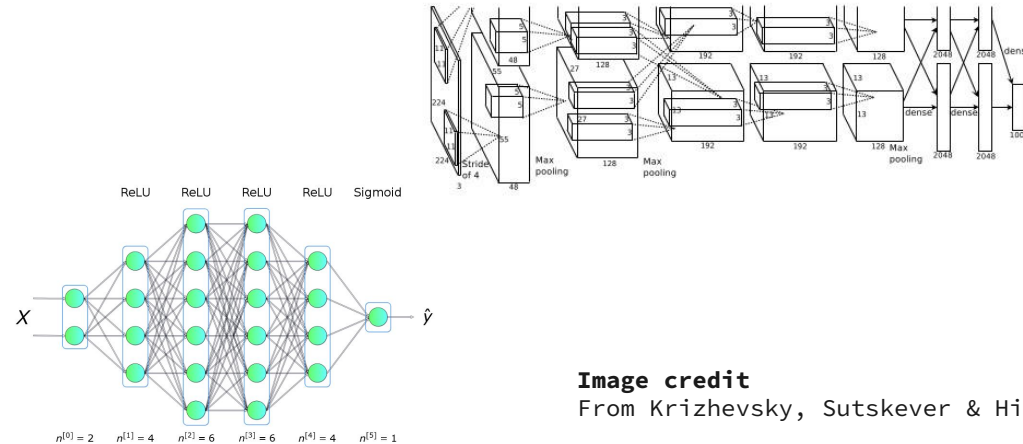


Image credit

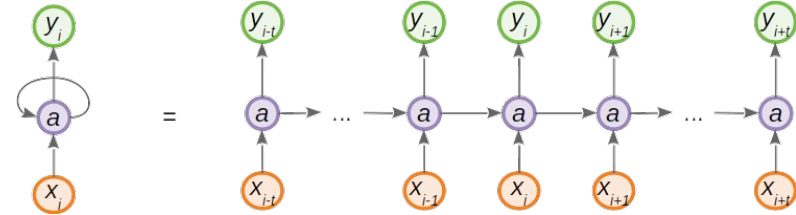
From Krizhevsky, Sutskever & Hinton (2012)

Recurrent networks

Variable size inputs

Popular for NLP & time series prediction

E.g. RNN, LSTM & GRU



Class 9



NN architectures

- **Perceptron** (lecture 2)
- **Multi-Layer Perceptron (MLP)** (lecture 2)
- **Convolutional Neural Network (CNN)** (lecture 4)
- **Graph Neural Network (GCN)** - a generalization of CNN)
- **Recurrent Neural Network (RNN)** (lecture 9)
- **Long-Short Term Memory (LSTM)** (lecture 9)
- **Deep Belief Network**
- **Autoencoder** (lecture 6)
- **Generative Adversarial Network (GAN)** (lecture 8)
- **Transformer** (lecture 10)
- & more ...

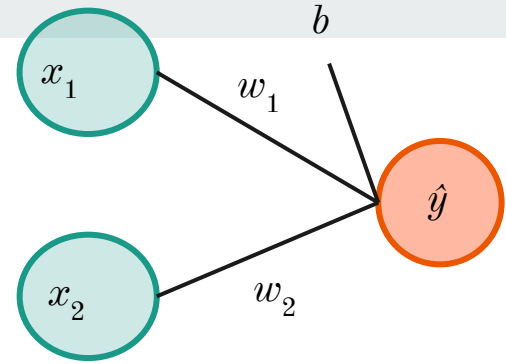
Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology & to scale such models to very large size.

For sequential data, another powerful specialization of the neural network framework.

The forward pass

- A simple network
- The forward pass
- The cycle of learning

A single neuron – the Perceptron



Two input (x_1 & x_2) example in the figure:

"Blue-orange classifier"

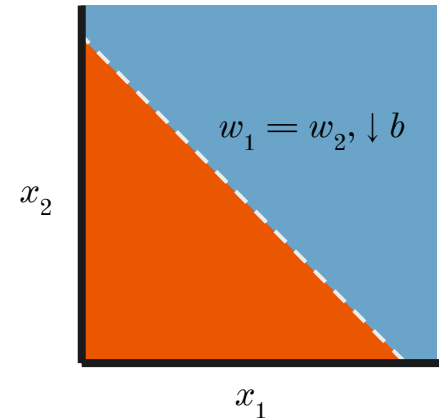
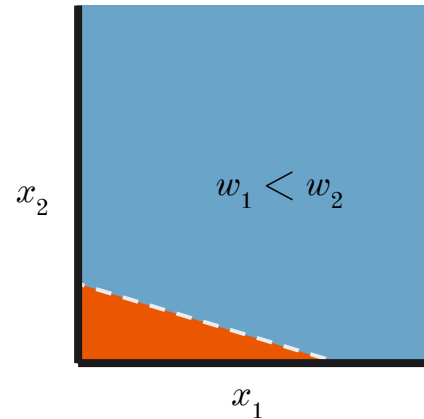
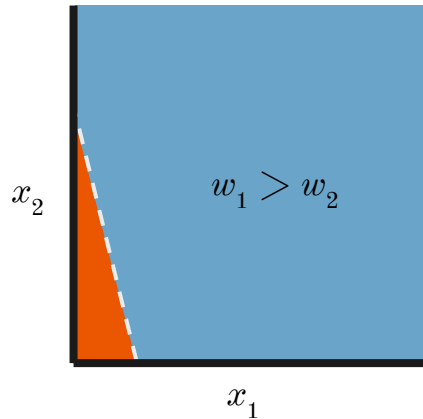
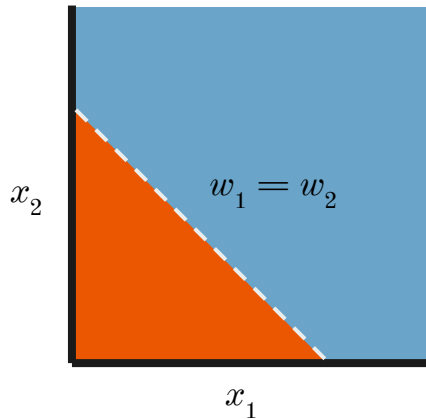
if $x_1 w_1 + x_2 w_2 + b > 0$:

blue

else:

orange

$$y_{\text{predict}} = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + b > 0 \\ 0 & \text{otherwise} \end{cases}$$



Question: Could a network like this, in principle, be used for a cat & dog classifier?

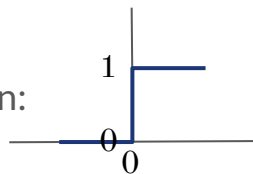
Question: Could a network like this, in principle, be used for an apple, orange & banana classifier?

A single neuron – logistic regression

Logistic regression is very similar to the Perceptron.

The only difference is the activation function.

Activation function in the Perceptron:



Activation function in the logistic regression:

$$f(x) = \frac{1}{1 + e^{-x}}$$

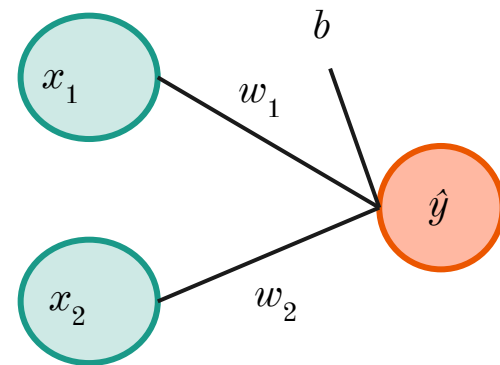
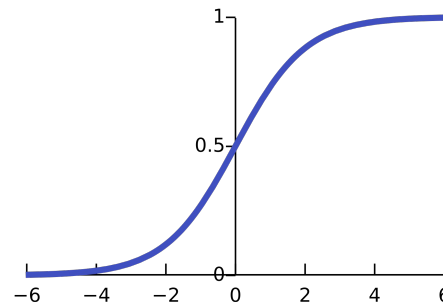


Image credit (sigmoid):

Qef - Created from scratch with gnuplot, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=4310325>

Question: Could a network like this, in principle, be used for an apple, orange & banana classifier?

A layer of neurons w/o activation function

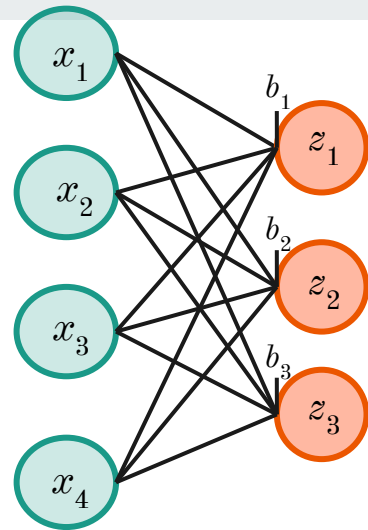
- The logits (Z) from matrix multiplication

$$\begin{aligned}z_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1 \\z_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2 \\z_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3\end{aligned}$$

$$\begin{array}{ccc|c} z_1 & z_2 & z_3 & \\ \hline & Z & & \end{array} = \begin{array}{ccc|c} x_1 & x_2 & x_3 & x_4 \\ \hline & X & & \end{array} \times \begin{array}{|c|c|c|} \hline w_{11} & w_{12} & w_{13} \\ \hline w_{21} & w_{22} & w_{23} \\ \hline w_{31} & w_{32} & w_{33} \\ \hline w_{41} & w_{42} & w_{43} \\ \hline & W & & \end{array} + \begin{array}{|c|c|c|} \hline b_1 & b_2 & b_3 \\ \hline & B & & \end{array}$$

$$Z = XW + B$$

$$Z = \text{np.dot}(X, W) + B$$



Step 1

Forward propagation

- generates the prediction

Inputs

$$x_1, x_2, x_3$$

Weights/parameters

$$w_{11}^1, w_{12}^2, \dots$$

Activation functions

$$f(\dots), g(\dots)$$

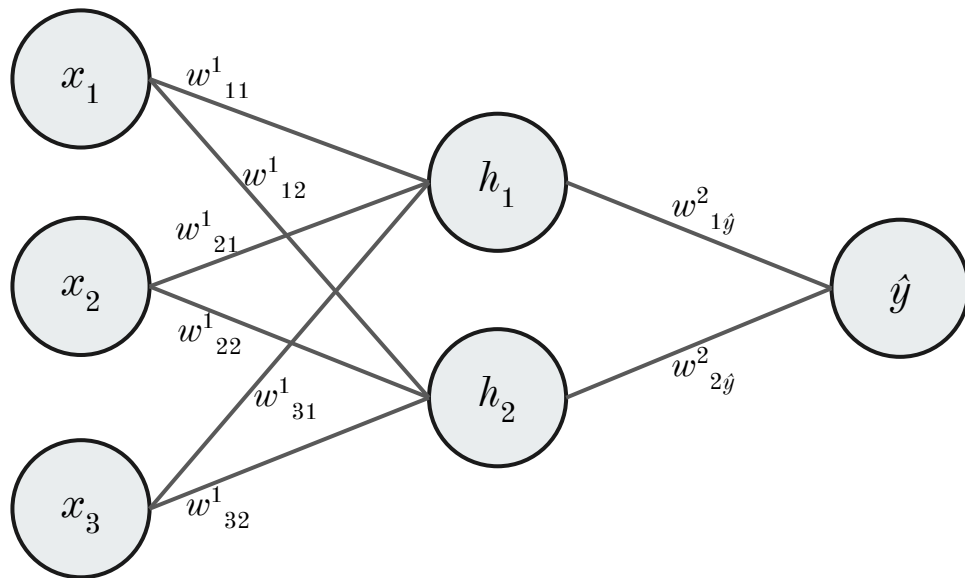
Activations

$$h_1, h_2$$

Output

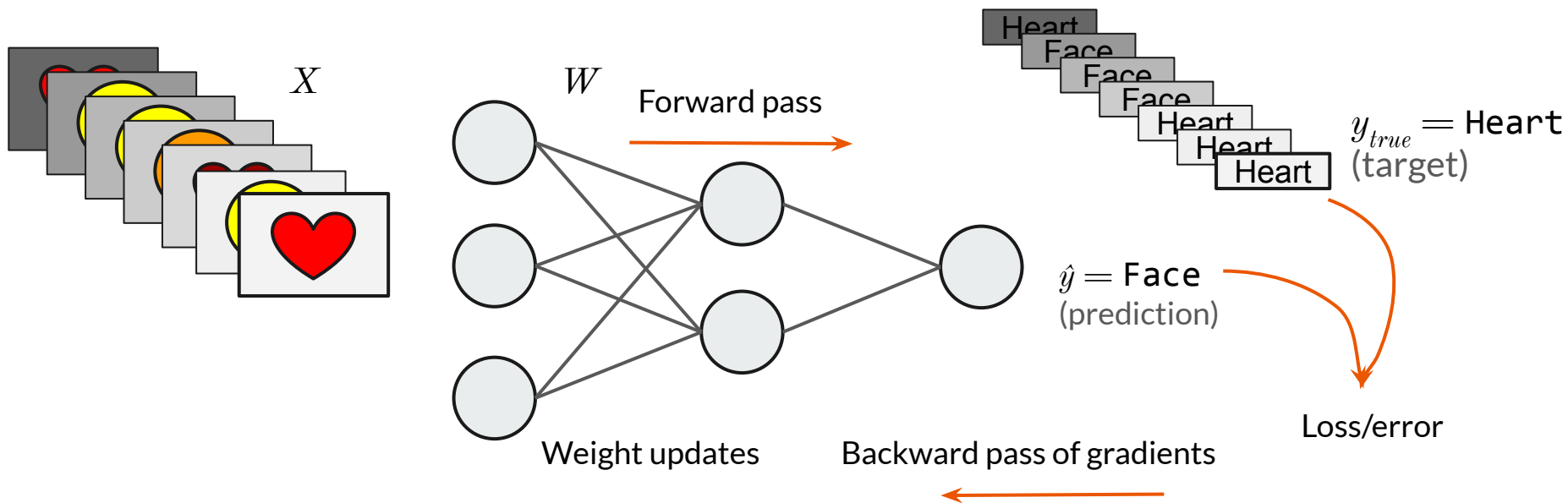
$$\hat{y}$$

$$\begin{aligned} z_1^1 &= w_{11}^1 x_1 + w_{21}^1 x_2 + w_{31}^1 x_3 \\ h_1 &= f(z_1^1) \\ h_2 &= f(w_{12}^1 x_1 + w_{22}^1 x_2 + w_{32}^1 x_3) \\ \hat{y} &= g(w_{1y}^2 h_1 + w_{2y}^2 h_2) \end{aligned}$$



The cycle of learning

1. Forward pass & prediction
2. Error/loss
3. Backward pass of error gradients
4. Weight updates



Activation functions

- The problem: fitting a non-linearity
- Multi-layer neural networks
- Activation functions for hidden neurons
- Example: fitting a sinusoid
- Neural networks are universal function approximators
- Activation functions for output neurons

A single layer

- what it **cannot** do

A linear classifier cannot do
non-linear classification

Example: xor (exclusive or)

XOR

Input

$$x_1 = 1, x_1 = 0$$

$$x_2 = 0, x_2 = 1$$

Output

$$y = 1$$

OR

Input

$$x_1 = 1, x_1 = 0, x_1 = 1$$

$$x_2 = 1, x_2 = 1, x_2 = 0$$

Output

$$y = 1$$

AND

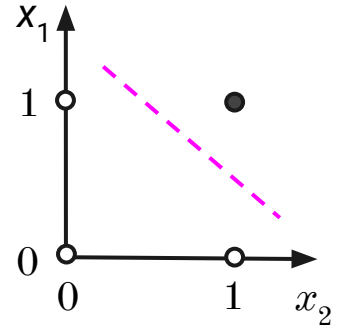
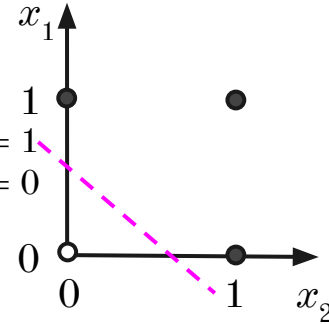
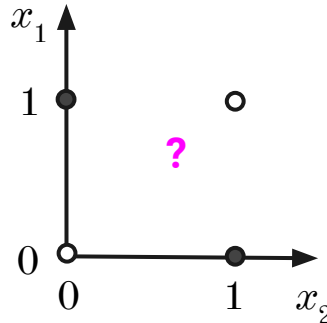
Input

$$x_1 = 1$$

$$x_2 = 1$$

Output

$$y = 1$$



**But... it is possible
with multiple layers**

Mcculloch & Pitts (1943)
showed that, for example, XOR
classification could be achieved
with multiple layers.

How to learn the weights in
multi-layer networks?

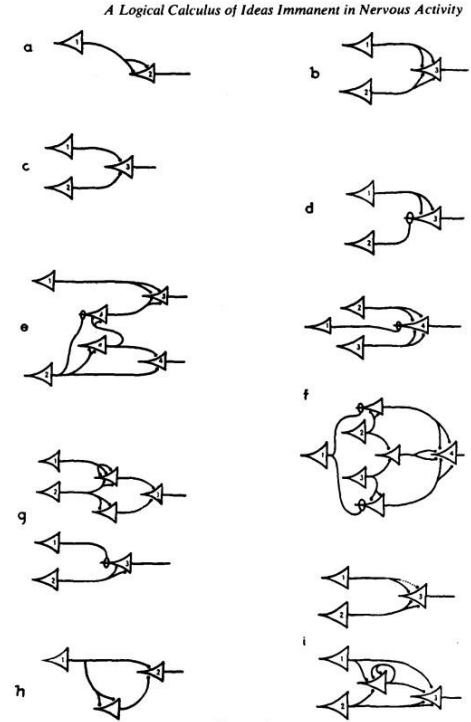
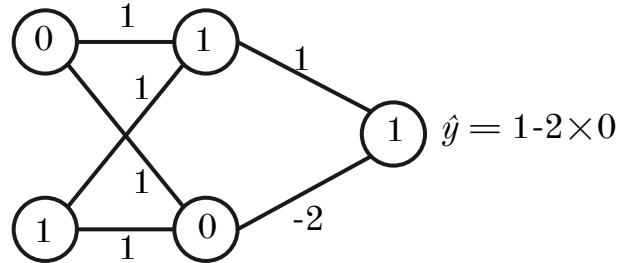
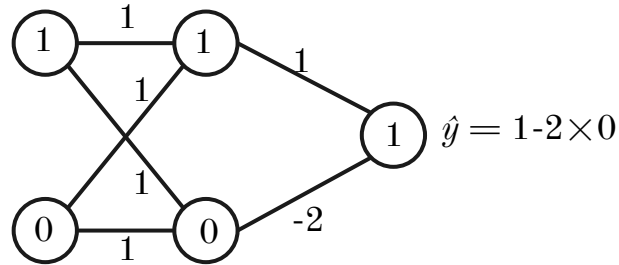
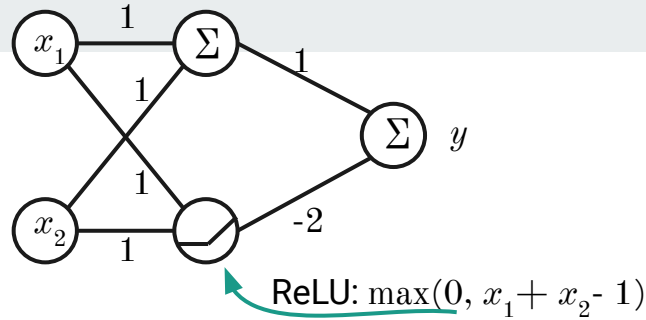


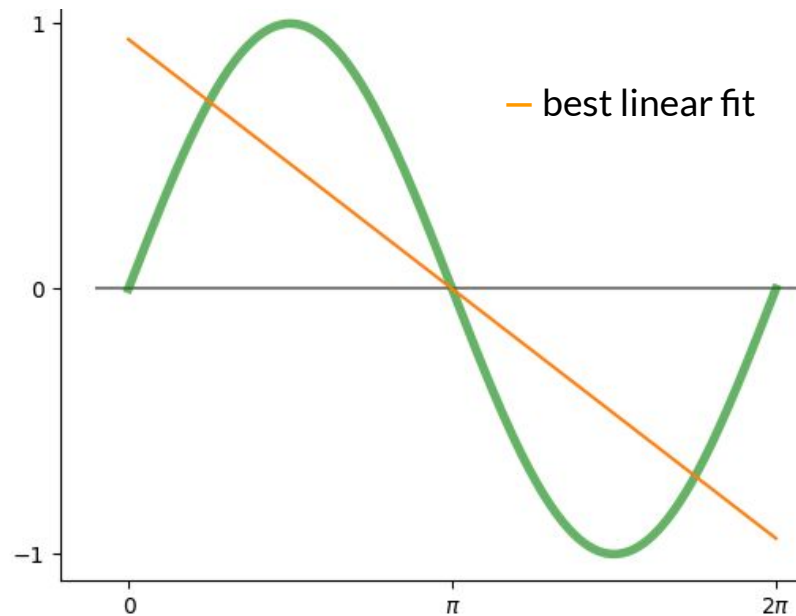
FIGURE 1

Question: Can we make the fit better by adding more linear neurons, or more layers of linear neurons?

Regression

- Example: fitting a sine wave

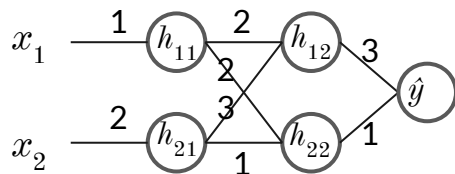
Example: a fit to a sine wave (from 0 to 2π) with a neural network with only linear units.



Why do multi-layer neural networks need non-linear activation functions?

Example:

- Two layer network with *linear* activations.
- *This is equivalent to a single layer network.*



Hidden layer 1

$$h_{11} = 1x_1$$
$$h_{21} = 2x_2$$

Hidden layer 2

$$h_{12} = 2h_{11} + 3h_{21}$$
$$h_{22} = 1h_{21} + 2h_{11}$$

Output layer

$$\hat{y} = 3h_{12} + 1h_{22}$$
$$\hat{y} = x_1(1 \times 2) + x_2(2 \times 3 \times 3) + x_2(2 \times 1) + x_1(1 \times 2 \times 1)$$
$$\hat{y} = x_1w_a + x_1w_b + x_2w_c + x_2w_d$$
$$\hat{y} = x_1(w_a + w_b) + x_2(w_c + w_d)$$

w_e w_f

A single layer



Why do neural networks have multiple layers?

Multiple layers with *linear* activations can be reduced to a single layer.

That is, multi-layer neural networks with *linear* activations are equivalent to logistic regression.

Question: Why is this a problem?

Activation functions for hidden neurons

Non-linear & differentiable

Represent a (\approx any) nonlinear function (given hidden layers)

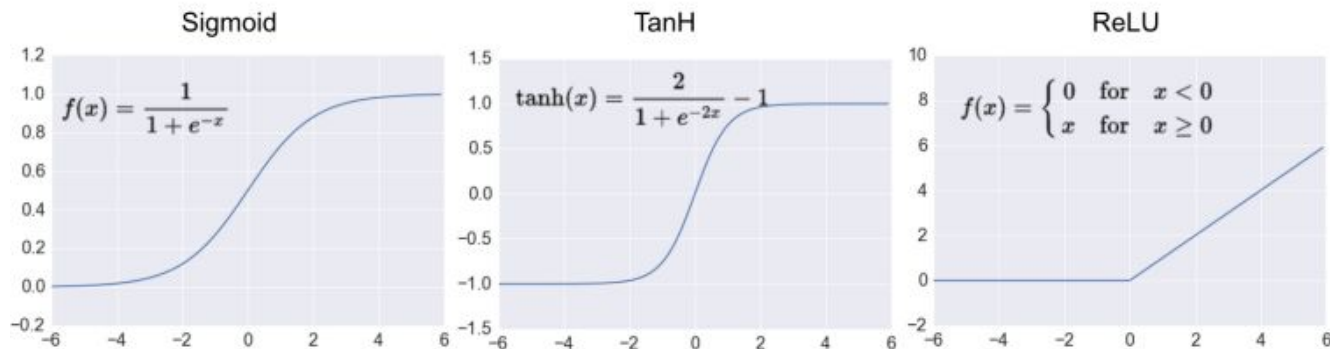
Allows for learning with error backpropagation

Popular choices

Sigmoid

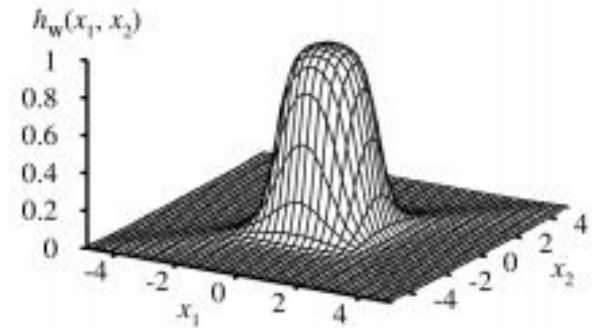
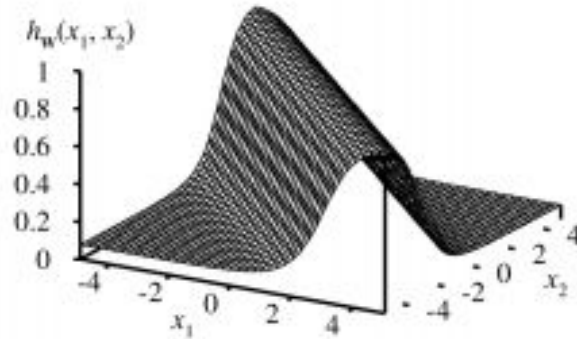
Hyperbolic tangent (tanh)

Rectified Linear Unit (ReLU): $\max(0, x)$



Activation functions for hidden layers

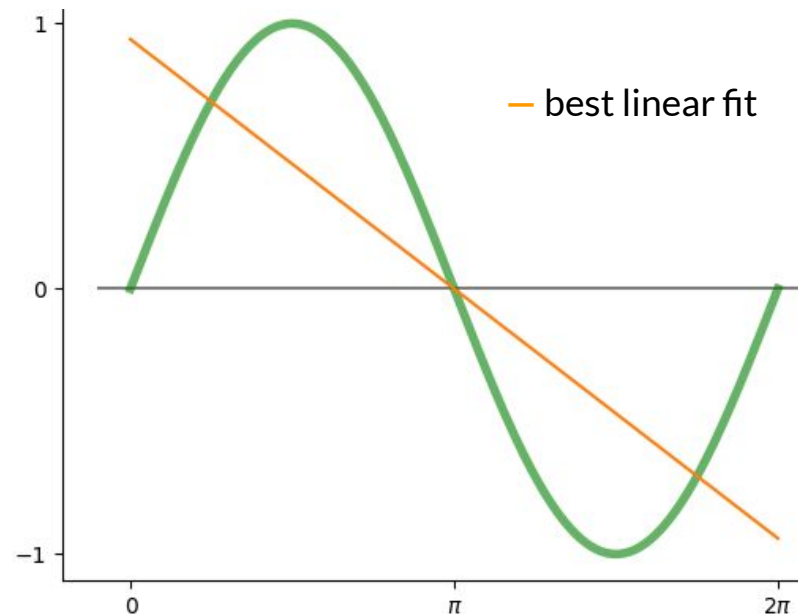
By combining non-linear activation functions increasingly complex “filters” can be created.

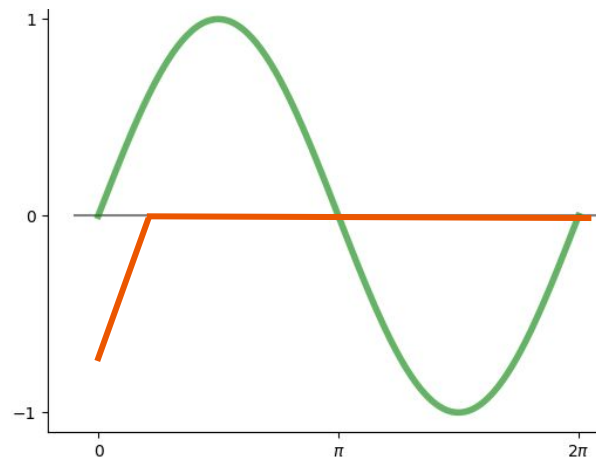
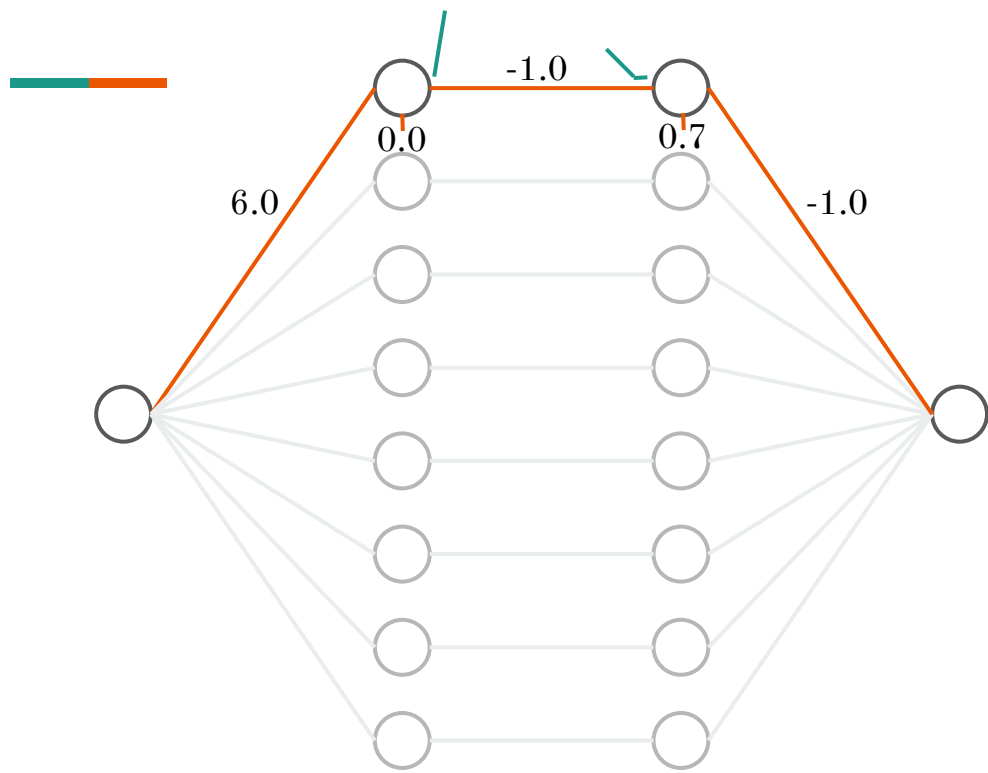


Fitting a sine wave

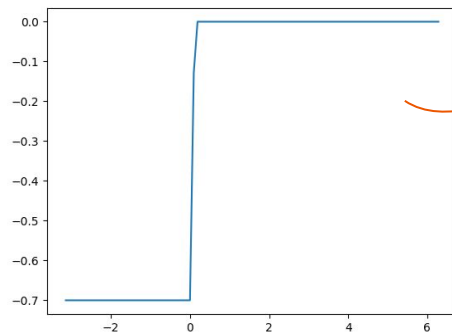
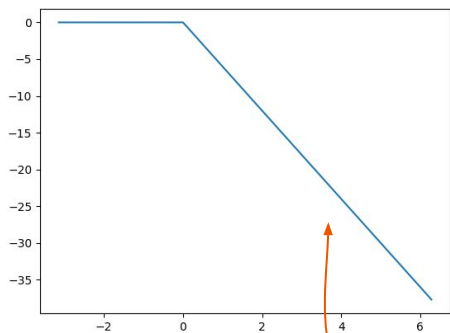
- Example continued

Example: a fit to a sine wave (from 0 to 2π) with a neural network with only linear units.

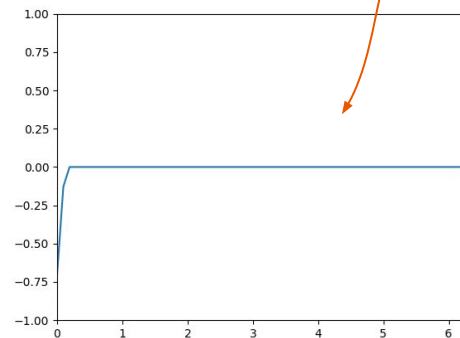
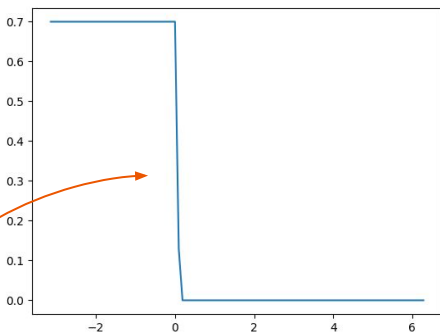
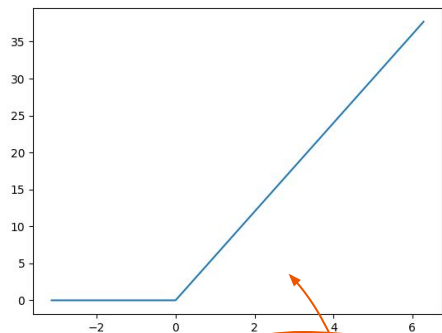




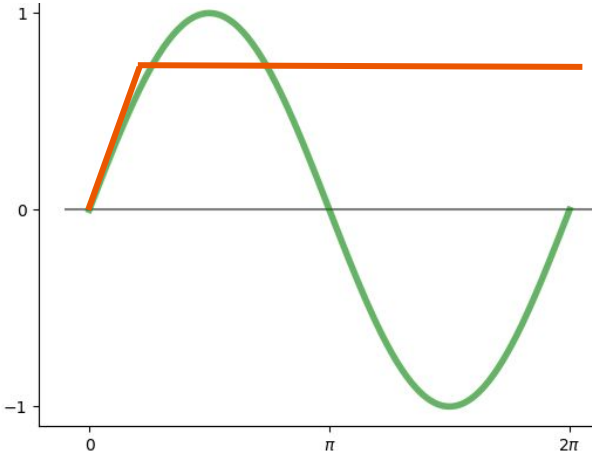
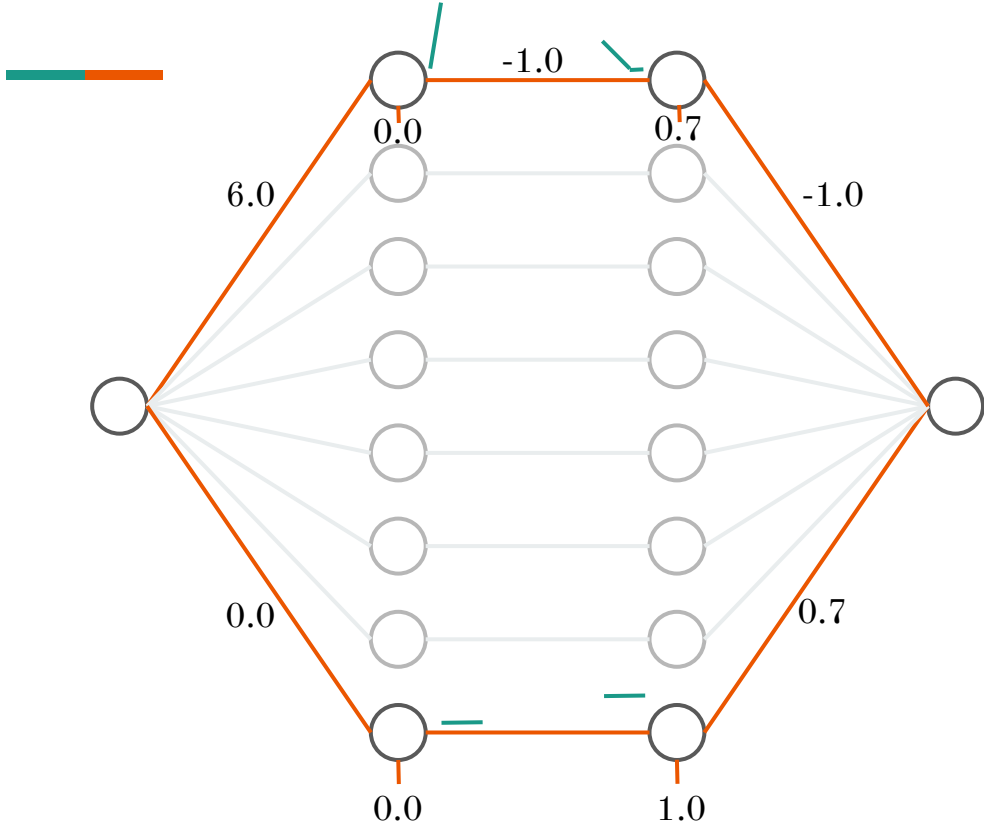
$$y = \max(\max(x \times 6.0 + 0.0, 0.0) \times (-1.0) + 0.7, 0.0) \times (-1.0)$$

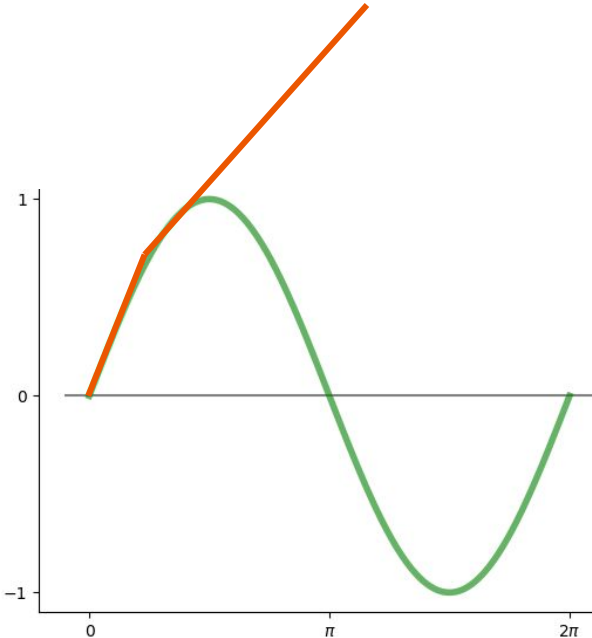
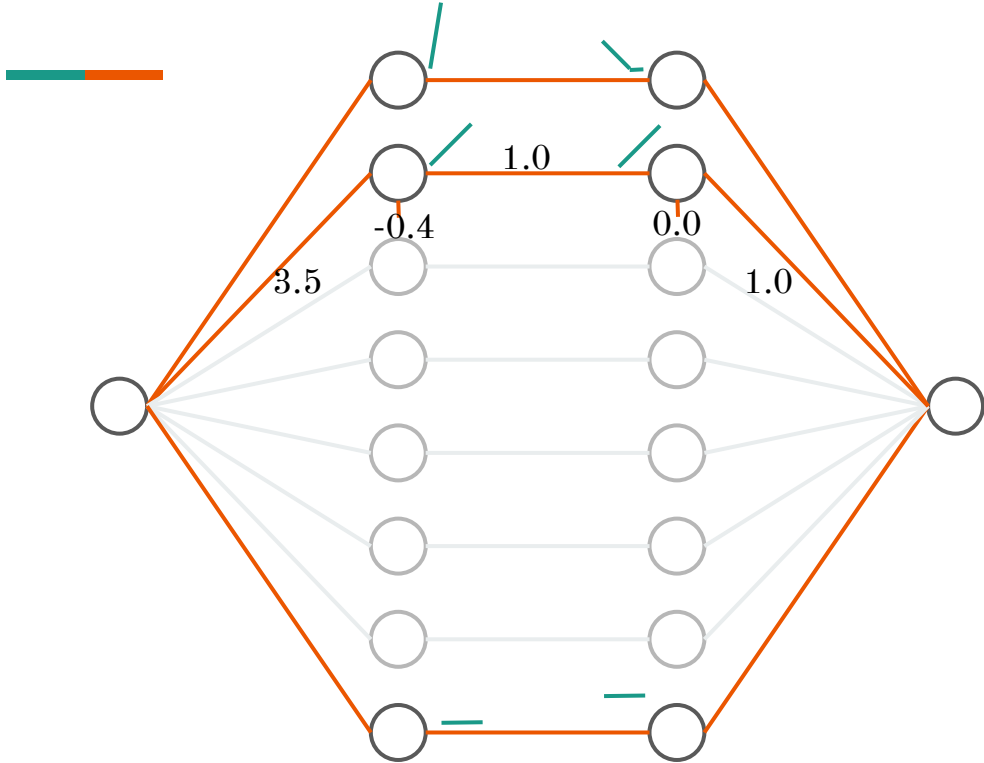


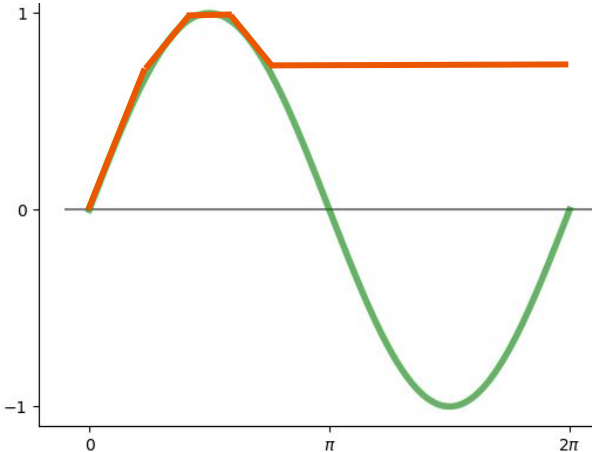
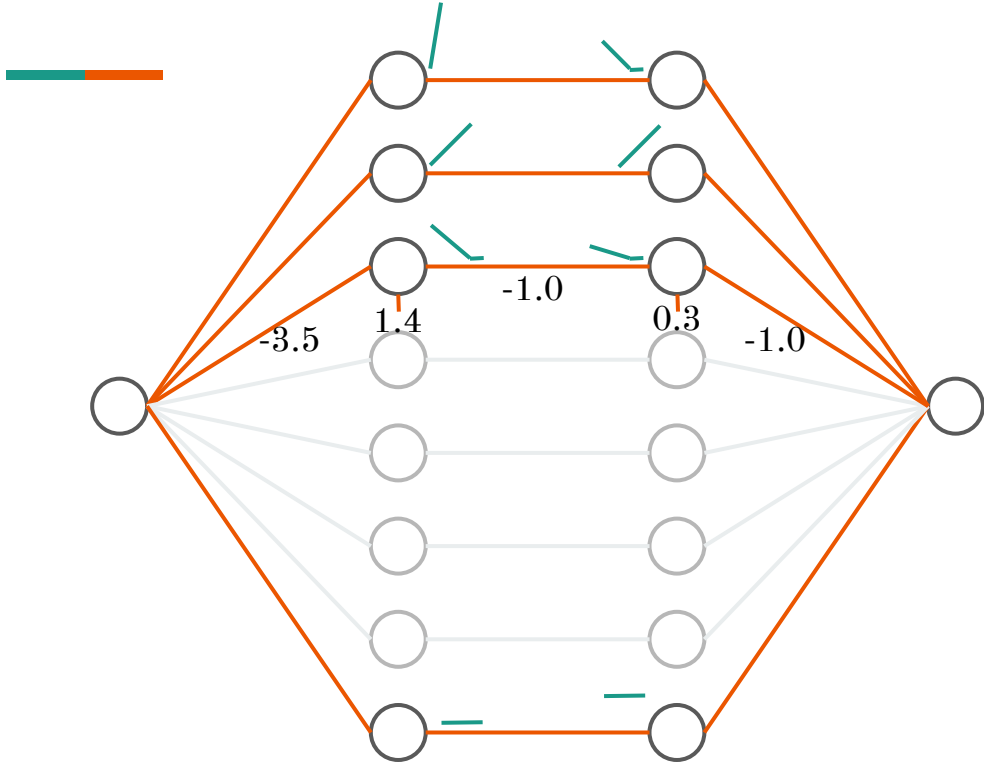
Set x & y limits

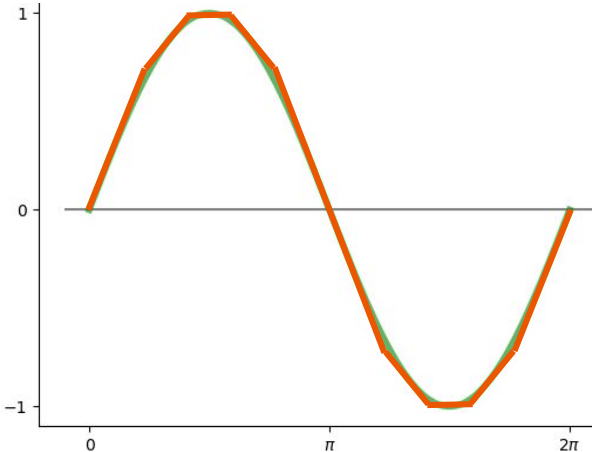
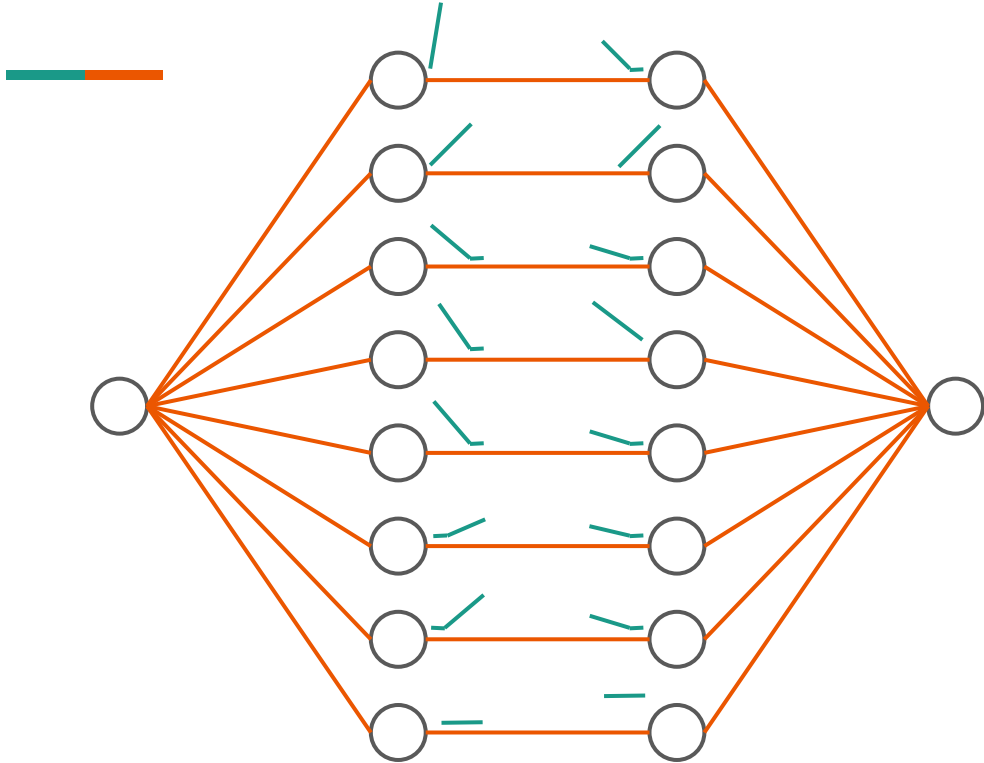


$$y = \max(\max(x \times 6.0 + 0.0, 0.0) \times (-1.0) + 0.7, 0.0) \times (-1.0)$$











Neural networks

- Universal function approximators

A feed-forward network with a single hidden layer can approximate all* continuous functions.

E.g.:

Cybenko, G. (1989)

Kurt Hornik (1991)

I.e., simple neural networks can represent a wide variety of interesting functions when given appropriate parameters.

* in compact subsets of \mathbb{R}^n , i.e. closed & bounded sets of real numbers

Question: Do activation functions for output layers have to be non-linear?

Question: What are the *logits*?

Activation functions for output layers

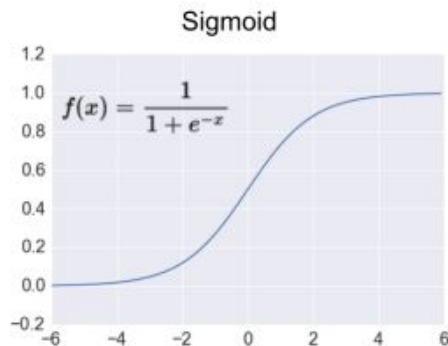
These also need to be differentiable.

Regression

- Linear activation

Binary classification

- Sigmoid function



Multi-class classification

- Softmax

```
e_z = np.exp(z - np.max(z, axis=1, keepdims=True))  
probs = e_z / e_z.sum(axis=1, keepdims=True)
```

$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

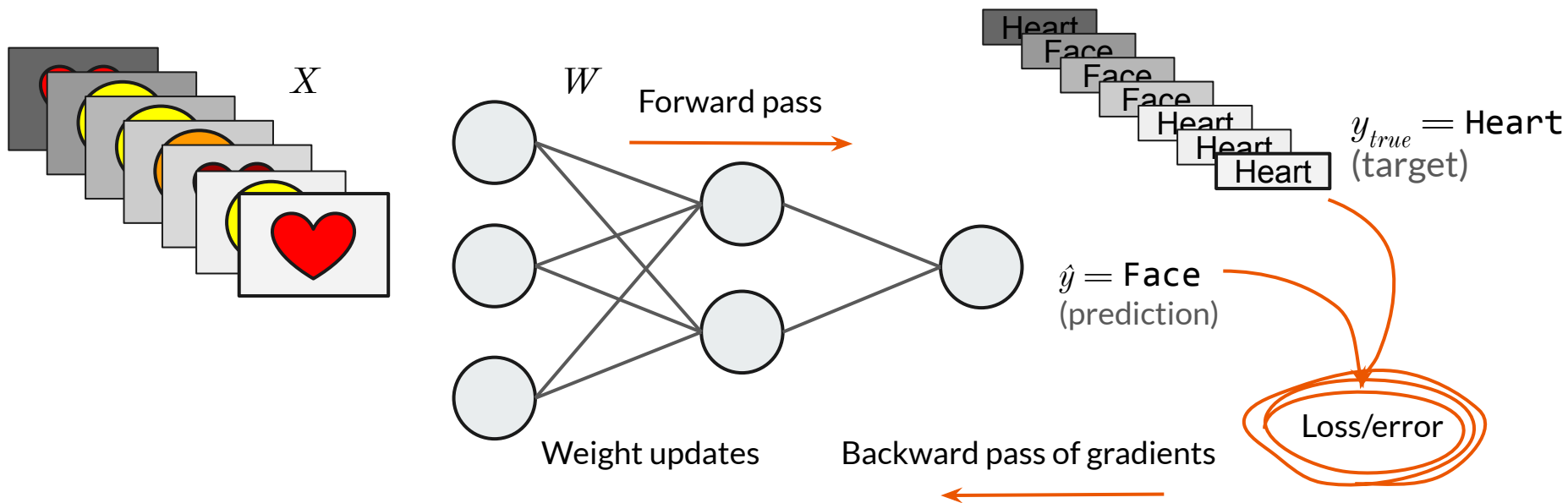
For an apple, orange & banana classifier, what is the max value of j ?

z_i is the logit for class i & j indexes all classes.

Loss function & target encoding

The cycle of learning

1. Forward pass & prediction
2. Error/loss
3. Backward pass of error gradients
4. Weight updates



Loss functions

The *error* is often called **loss** or **cost**

3 common cases:

- **Regression**

Mean Squared Error (MSE)

- **Binary classification**

Binary Cross-Entropy

$$L = -y_{true} \log(y_{predict}) + (1 - y_{true}) \log(1 - y_{predict})$$

- **Multi-class classification**

Cross-Entropy

$$L = - \sum_{k=1}^K y_{true}^k \log(y_{predict}^k)$$

Real numbers

Binary (0 or 1)

Real between 0 & 1

$$L = \frac{1}{m} \sum_{i=1}^m (y_{true}^i - y_{predict}^i)^2$$

Question: What do probs correspond to in the mathematical expression?

Question: What do oh_y_true correspond to in the mathematical expression?

Question: Why are probs clipped?

Loss functions

Cross-Entropy (aka Categorical Cross-Entropy)

$$L = - \sum_{k=1}^K y_{true}^k \log(y_{predict}^k)$$

Softmax output (probability distribution over classes)

One-hot encoded categories

Python & NumPy

```
probs_clipped = np.clip(probs, 1e-7, 1 - 1e-7)
loss = -np.sum(oh_y_true * np.log(probs_clipped), axis=1).mean(axis=0)
```




One hot encoding

- How to compare apples to oranges

Loss

- Mean squared error:

$$L = \frac{1}{m} \sum_{i=1}^m (y_{true}^i - y_{predict}^i)^2$$

Regression (simple)

- Example:

$$y_{true}^i = 23.5; y_{predict}^i = 22.9$$

$$L = (23.5 - 22.9)^2$$

$$L = 0.36$$

One hot encoding

- How to compare apples to oranges

Coding of categorical, not ordered values

- *One-hot encoding*

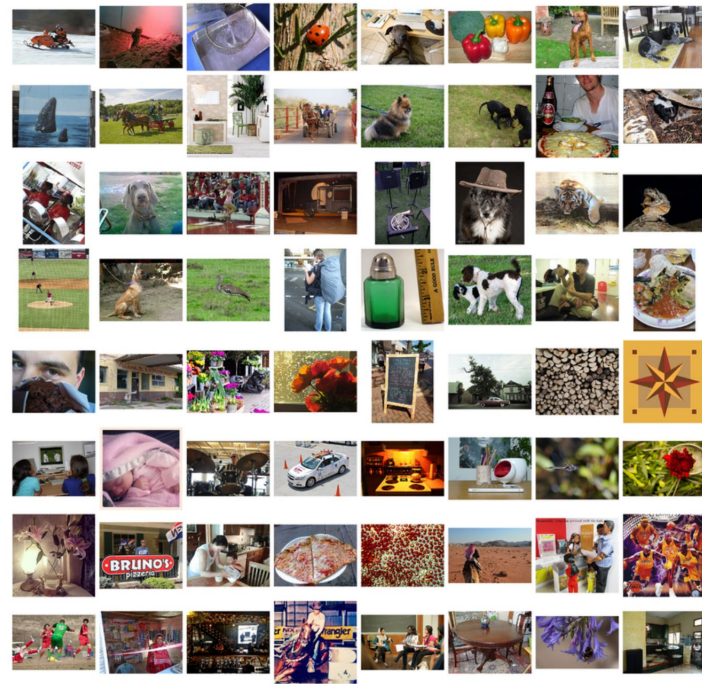
- Example:

Dog: [1, 0, 0, 0]; Pear: [0, 1, 0, 0];

House: [0, 0, 1, 0]; Grass: [0, 0, 0, 1]

- A sparse encoding

Most of the values are zero



One hot encoding

- How to compare apples to oranges

Coding of categorical, not ordered values

- **One-hot encoding**

Example:

Dog: [1, 0, 0, 0]; Pear: [0, 1, 0, 0];

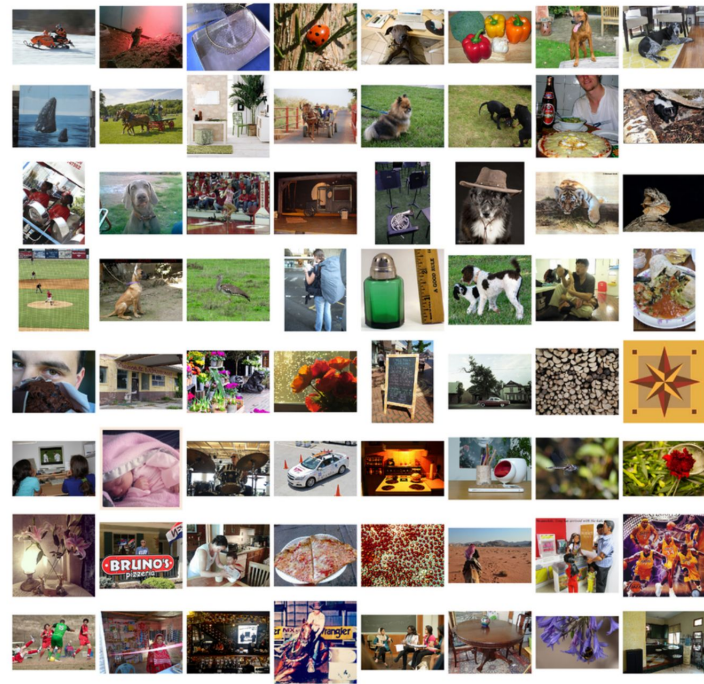
House: $[0, 0, 1, 0]$; Grass: $[0, 0, 0, 1]$

- MSE loss & one-hot encoding

$$L = \sum ([0, 1, 0, 0] - [0, 0, 1, 0])^2; L = 2$$

$$L = \sum ([1, 0, 0, 0] - [0, 0, 0, 1])^2; L = 2$$

Just an illustration, MSE is not actually used with one hot encoding



Question: What is \hat{y} ?

One hot encoding

- Predictions (\hat{y} aka $y_{predict}$)

Predicting categorical, not ordered values

- **Softmax** $y_{predict} = \frac{e^z}{\sum_{k=1}^K e^{z_k}}$

Returns the probability that the input belongs to a given class

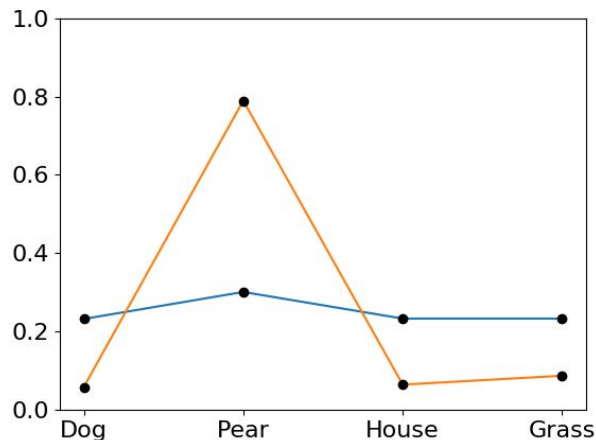
An output per class (a $y_{predict}$ per class)

I.e. a probability distribution over the classes

- Example predictions

blue: [0.232, 0.301, 0.233, 0.233]

yellow: [0.059, 0.789, 0.065, 0.087]



Question: What is this loss function called?

One hot encoding

- Loss

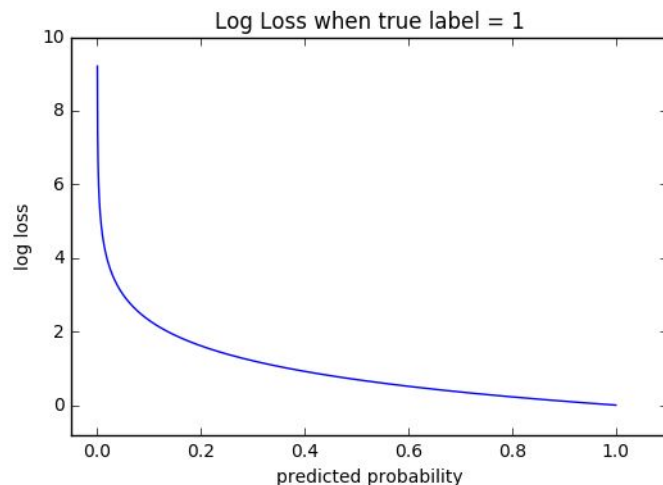
Loss

Cross-entropy/log loss (not MSE) for $K > 2$:

$$L = - \sum_{k=1}^K y_{true}^k \log(y_{predict}^k)$$

for $k = 2$:

$$L = -(y_{true} \log(\hat{y}) + (1 - y_{true}) \log(1 - \hat{y}))$$



$$y_{predict} = \frac{e^z}{\sum_{k=1}^K e^{z_k}}$$

Question: What output reflect greater confidence?

One hot encoding

- Loss & confidence

Cross-entropy/log loss (not MSE) for $K > 2$

Example

- True class: Pear [0, 1, 0, 0]
- Predictions:

blue: [0.23, 0.30, 0.23, 0.23]

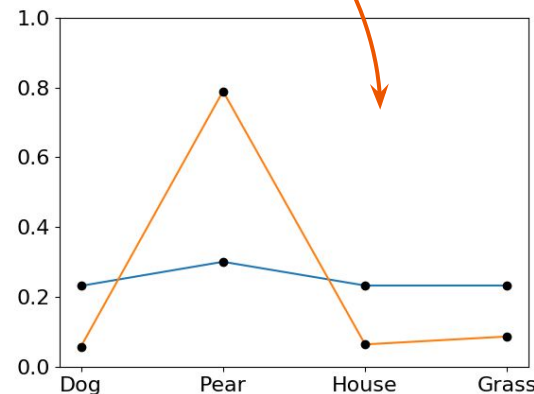
yellow: [0.06, 0.79, 0.07, 0.09]

- Loss:

blue: $-(([0., 1., 0., 0.] * \text{np.log}([0.23, 0.30, 0.23, 0.23]))). \text{sum}() = 1.20$

yellow: $-(([0., 1., 0., 0.] * \text{np.log}([0.06, 0.79, 0.07, 0.09]))). \text{sum}() = 0.24$

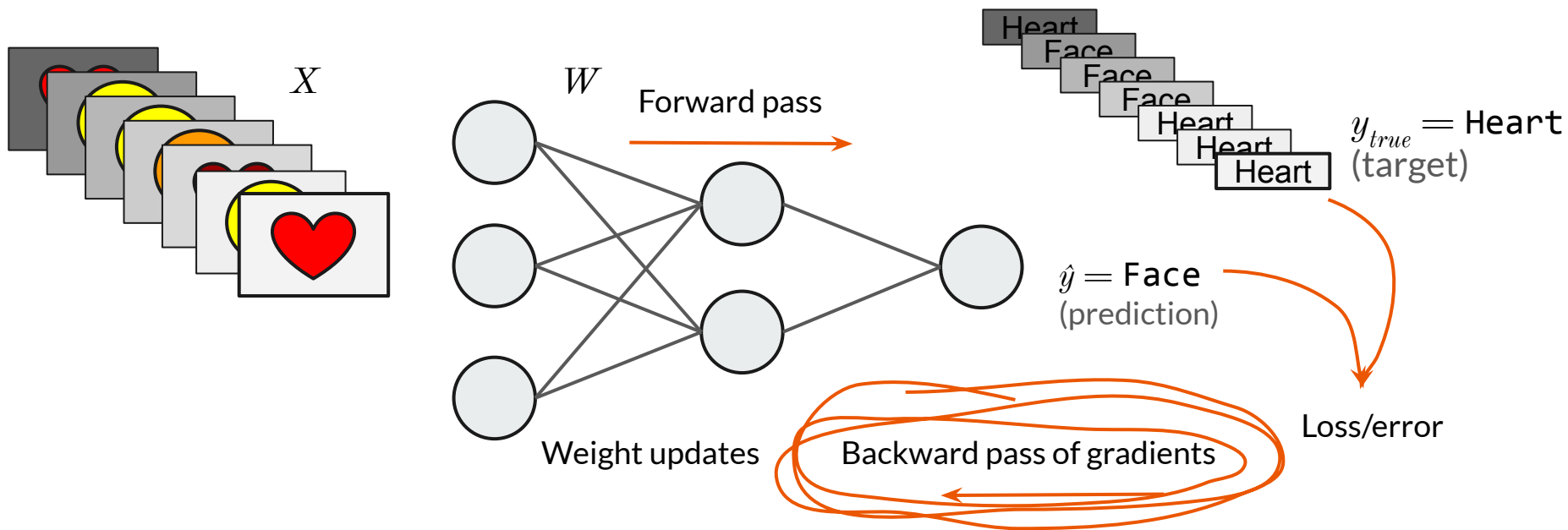
Softmax, one-hot encoding & cross-entropy captures the model's **confidence** in a prediction



The backward pass

The cycle of learning

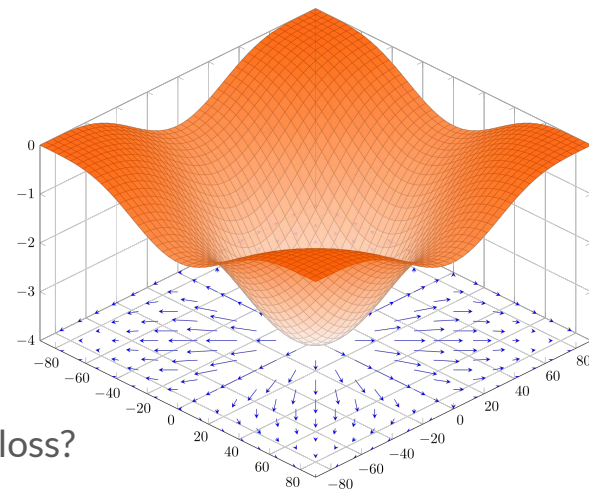
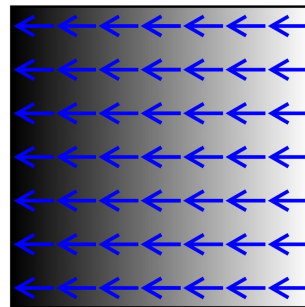
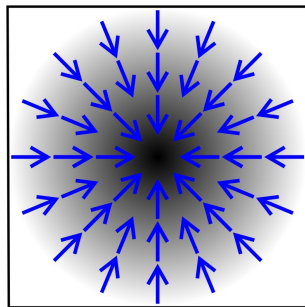
1. Forward pass & prediction
2. Error/loss
3. Backward pass of error gradients
4. Weight updates



Question: What is a *partial derivative*?

Question: What does darker grey indicate?

Gradients



At the end of the forward pass we got a *loss*.

How do we *change* the parameters (weights & biases) to *minimize* this loss?

- Partial derivative of loss with respect to the parameters.
- This tells us how the loss changes (increases or decreases) when we change a parameter.

Gradient $\nabla f(x, w) = \begin{bmatrix} \frac{\partial E}{\partial y} \\ \dots \\ \frac{\partial E}{\partial w_{21}^1} \end{bmatrix}$

- A vector of partial derivatives of the loss with respect to the parameters. The gradient points towards higher loss.

Image credits:

MartinThoma - CC0, <https://commons.wikimedia.org/w/index.php?curid=71375503>;

CC BY-SA 2.5, <https://commons.wikimedia.org/w/index.php?curid=1146042>

Question: What does i denote?

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

Kronecker delta

Derivatives of activation functions

ReLU

$$f(x) = \max(x, 0)$$

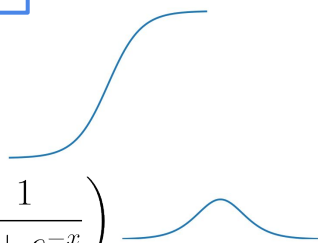
$$f'(x) = 1(x > 0)$$



Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = \frac{1}{1 + e^{-x}} \cdot \left(1 - \frac{1}{1 + e^{-x}}\right)$$



Softmax

$$S_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

$$\frac{\partial S_i}{\partial z_{i,j}} = S_i \delta_{ij} - S_i S_j$$

flatten to a column vector

```
prob = prob.reshape(-1, 1) # prob == S  
dz = np.diagflat(prob) - np.dot(prob, prob.T)
```

How to calculate the gradient

- The chain rule from calculus

During the forward pass the input is sent through the first layer & then the activation function, then then next layer, next activation functions, & so on. I.e. the input is sent through a chain of functions.

Individual functions:

$$g(\dots) = w_2 f_1(w_1 x)$$

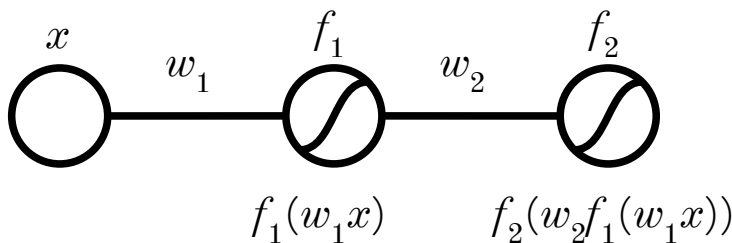
$$h(\dots) = f_2(w_2 f_1(w_1 x))$$

composite function:

$$F(\dots) = h(g(\dots))$$

derivative of composite function:

$$F'(\dots) = h'(g(\dots))g'(\dots)$$



(Re)discovered multiple times: Linnainmaa (1970), Werbos (1974), Rumelhart et al. (1986) ...



The gradient (again)

The gradient for a network is a vector containing all possible partial derivatives for that network.

$$\nabla f(x, w) = \begin{bmatrix} \frac{\partial E}{\partial y} \\ \dots \\ \frac{\partial E}{\partial w_{21}^1} \end{bmatrix}$$

Partial derivative of the loss
w.r.t. the predictions.

Partial derivative of the loss
w.r.t. weight w_{21}^1 .

Step 3 & 4

Question: What does \hat{y} mean?

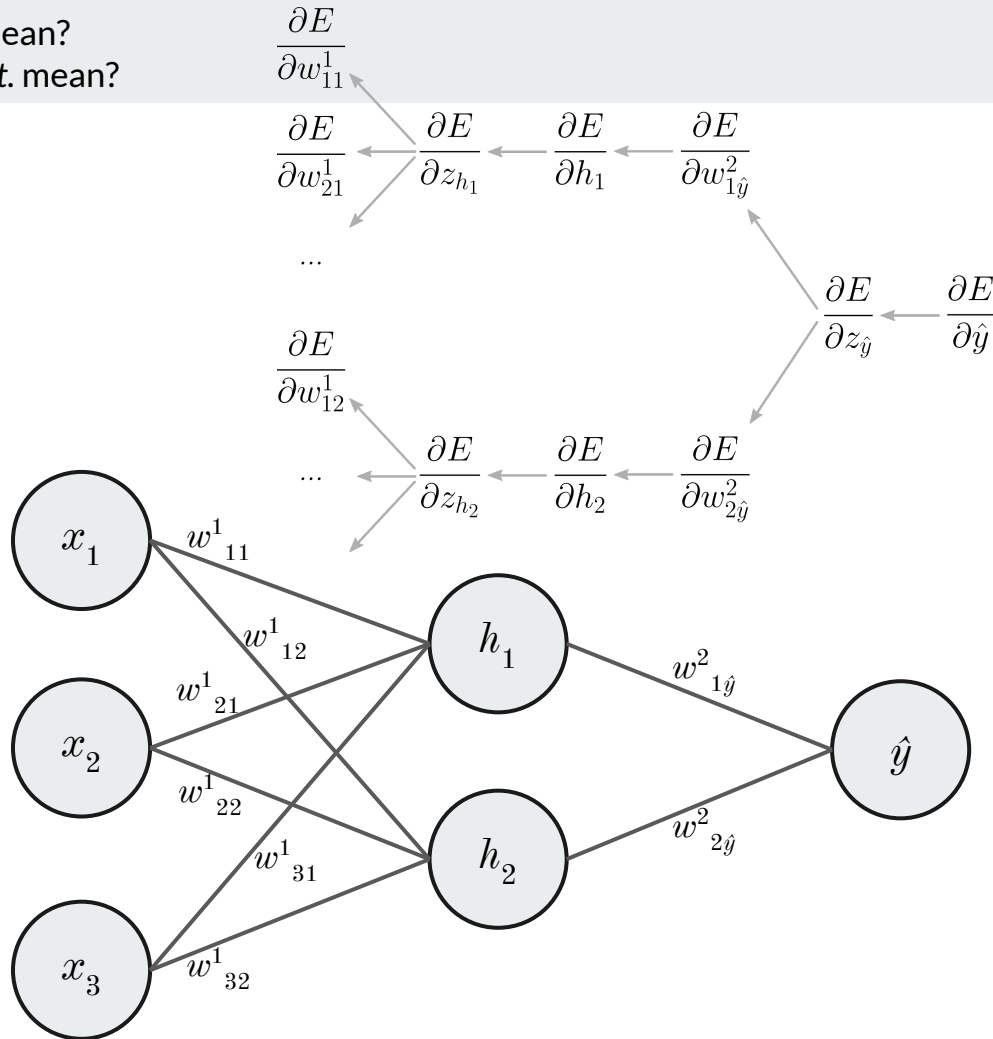
Question: What does w.r.t. mean?

Backprop

- backwards propagation of the error

1. Gradient of error w.r.t the output node (\hat{y})
2. Gradient of the error w.r.t. the input to \hat{y} ($z_{\hat{y}}$)
3. Gradient of the error w.r.t the weights ($w^2_{1\hat{y}}, w^2_{2\hat{y}}$)
4. Gradients of the error w.r.t. the output of the hidden layer nodes (h_1, h_2)
5. Gradients of the error w.r.t. the inputs to the hidden layer nodes (z_{h_1}, z_{h_2})
6. Gradient of the error w.r.t the weights ($w^1_{11}, w^1_{12}, w^1_{21}, w^1_{22}, w^1_{31}, w^1_{32}, \dots$)
7. Update all weights (W):

$$W^{t+1} = W^t - \alpha \frac{\partial E(X, W)}{\partial W}$$



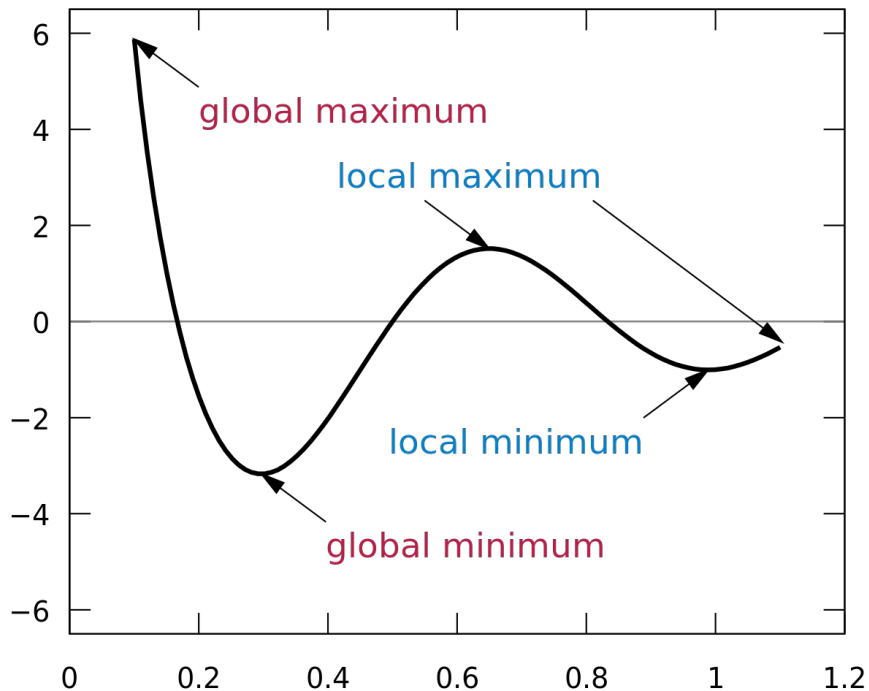
For a very nice visual illustration see

<https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/>

Local search & minima

Gradient descent is a form of local search

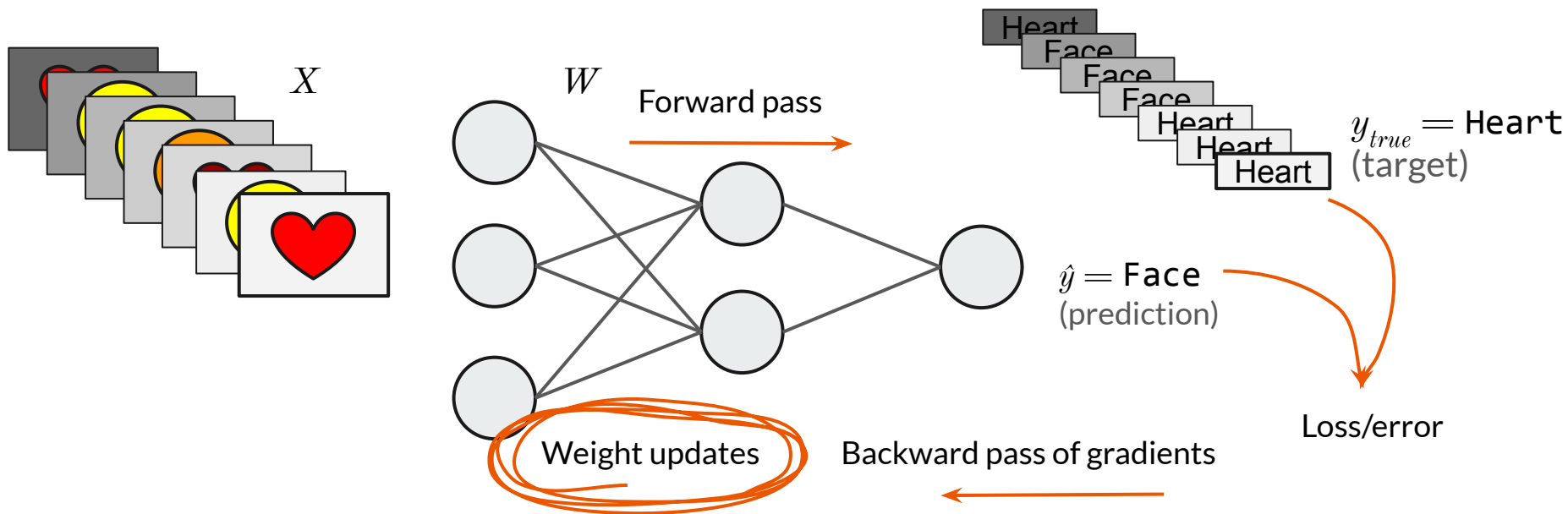
- it is not guaranteed to find the **global minimum**, but might get stuck in a **local minimum**




**Parameter update
aka optimization**

The cycle of learning

1. Forward pass & prediction
2. Error/loss
3. Backward pass of error gradients
4. Weight updates




$$W^{t+1} = W^t - \alpha \frac{\partial E(X, W)}{\partial W}$$

Stochastic Gradient Descent (SGD)

- The mother of all optimizers

Terminology

- **Stochastic gradient descent (SGD):** Historically this meant one update of the parameters per sample. Today, mini-batches are assumed & SGD refers to *mini-batch gradient descent*.
- **Batch gradient descent:** Historically this meant a single update of the parameters on the entire data set. I.e. the parameters are updated with the average gradient across the entire data set.
- **Mini-batch gradient descent:** The parameters are updated after each mini-batch (today simply called batch) of data. I.e. the average gradient is calculated over the mini-batch (e.g. 32 examples) & the parameters are updated.

Question: Does this implement single-sample SGD, mini-batch SGD or batch SGD?

Question: What are dweights & dbiases in the mathematical notation?

$$W^{t+1} = W^t - \alpha \frac{\partial E(X, \theta)}{\partial W}$$

$$b^{t+1} = b^t - \alpha \frac{\partial E(X, \theta)}{\partial b}$$



SGD with Python

```
weights = weights - learning_rate * dweights
```

```
biases = biases - learning_rate * dbiases
```

Or shorter yet:

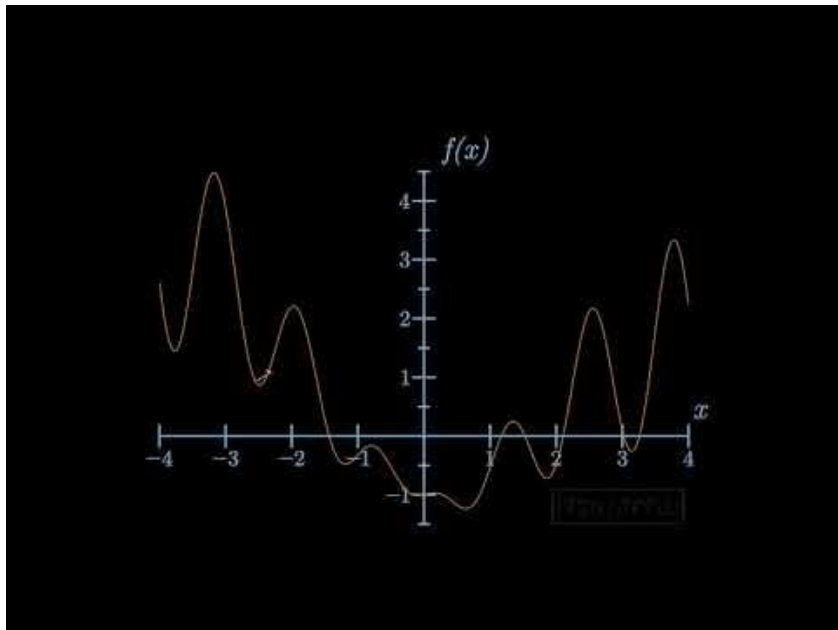
```
weights -= learning_rate * dweights
```

```
biases -= learning_rate * dbiases
```

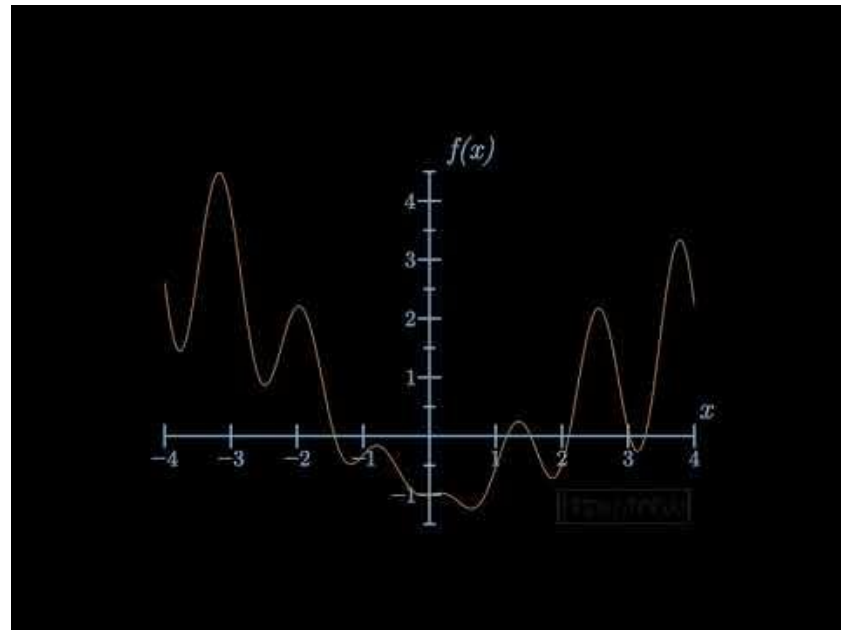
Where θ is all parameters, W is the weights & b is the biases.

Learning rate

Small learning rate

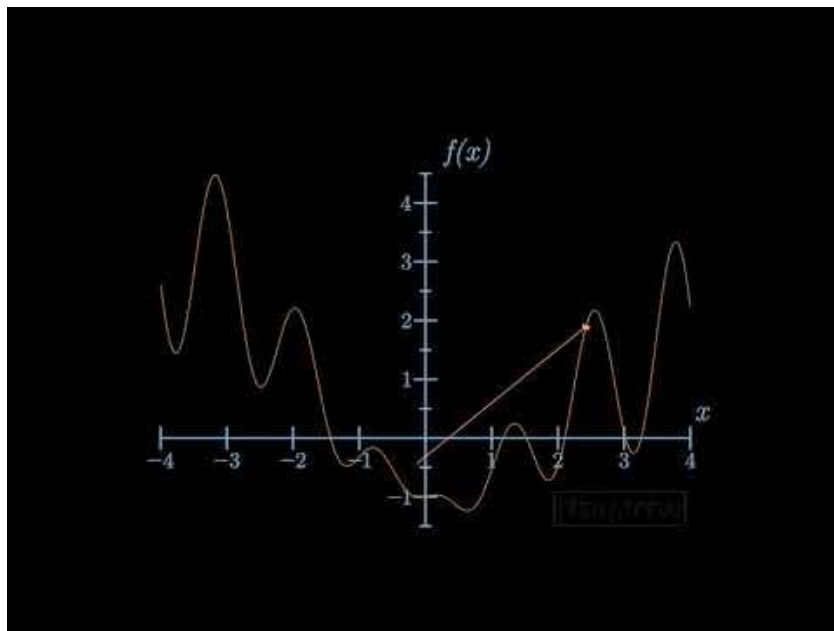


Better, but not good enough



Learning rate

Too big learning rate





Learning rate decay

Start out with a high learning rate & decrease it as training progresses.

```
init_learning_rate = 1.0
learning_rate_decay = 0.1
for step in range(20):
    learning_rate = init_learning_rate * (1. / (1 + learning_rate_decay * step))
```



Momentum

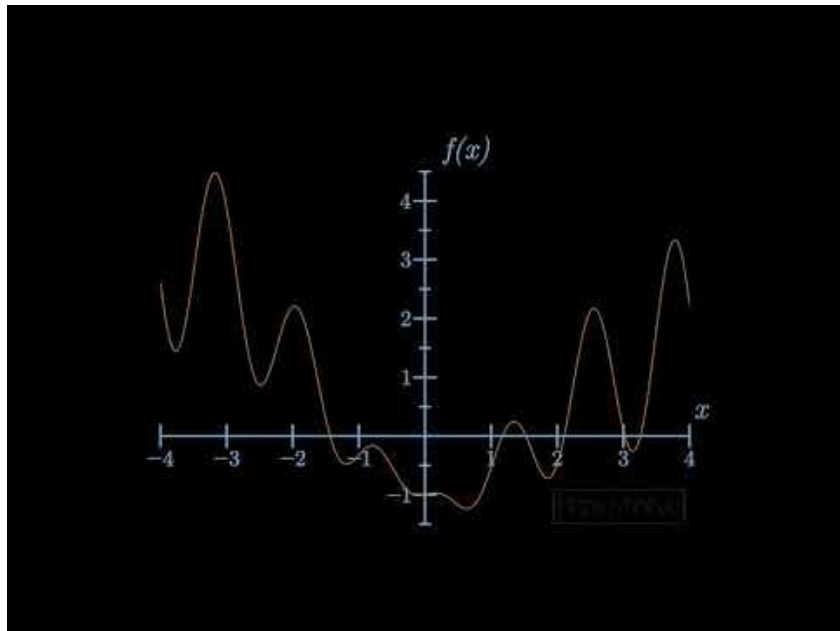
A moving average of gradients over some number of updates. This average acts like inertia, or a momentum, & is used together with the current gradient.

When the gradient points down over multiple consecutive updates the momentum increases & the updates continue in the direction of the momentum (like a ball rolling down a hill, picking up speed & then rolling over smaller obstacles).

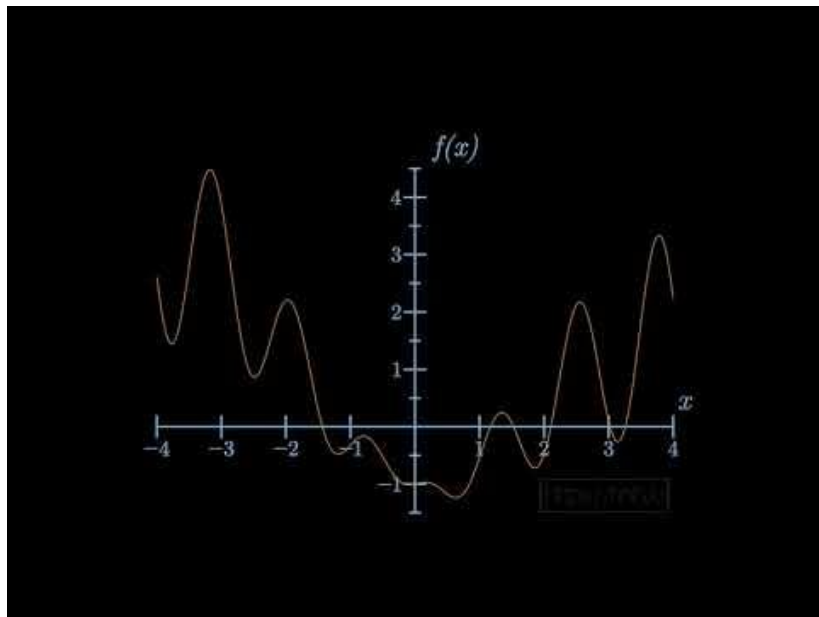
```
weight_updates = momentum * layer.weight_moms - curr_learning_rate * layer.dweights  
layer.weight_moms = weight_updates
```


Momentum & learning rate

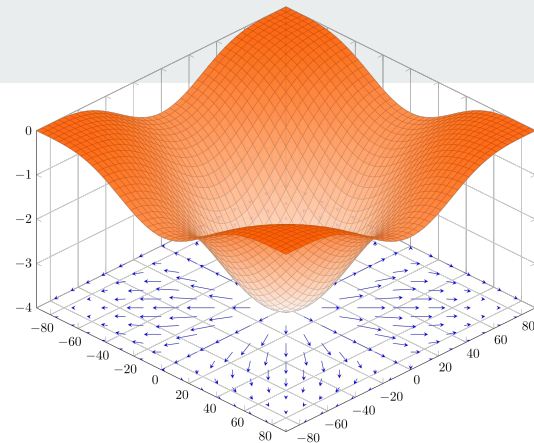
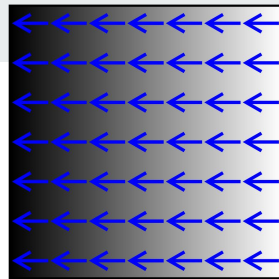
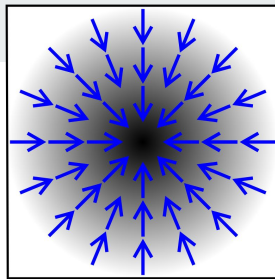
Large momentum & small learning rate



Good momentum & learning rate



Terminology



Gradient

$$\nabla f(x, w) = \begin{bmatrix} \frac{\partial E}{\partial y} \\ \dots \\ \frac{\partial E}{\partial w_{21}^1} \end{bmatrix}$$

- A vector of partial derivatives of the loss with respect to the parameters. The gradient points towards greater loss.

Gradient descent

- A first-order iterative optimization algorithm. Minimizing a function (e.g. a loss function) by stepwise moving opposite to the gradient.

Weight update

- Changing the weights in a direction opposite to the gradient & thus, taking a step towards a minimum.

Batch

- The examples over which the gradients are averaged for each weight update. E.g., if the batch size is 32 then the loss for 32 examples is computed before the weights are updated.

Image credits:

MartinThoma - CC0, <https://commons.wikimedia.org/w/index.php?curid=71375503>;

CC BY-SA 2.5, <https://commons.wikimedia.org/w/index.php?curid=1146042>

Question: What more terms should be included?

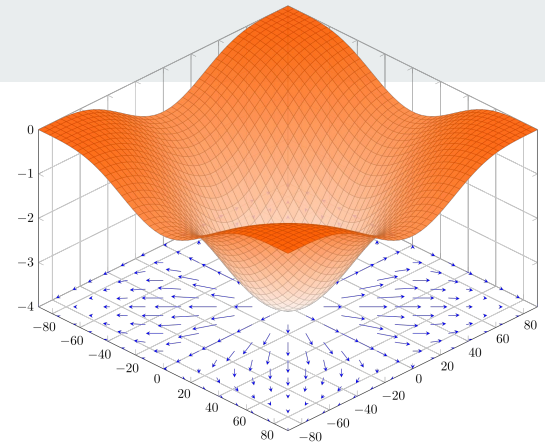
Terminology

Epoch

- A single pass through the entire training set

Stochastic gradient descent (SGD)

- Gradient descent with a batch size smaller than the size of the training set, so that the weights are updated multiple times per epoch.
- The correct term is actually mini-batch SGD.





For more

3Blue1Brown:

[But what is a neural network? | Chapter 1, Deep learning](#)



Tutorial

[MMAI5500_class02_NN.ipynb](#)

[MMAI5500_class02_learningrate.ipynb](#)