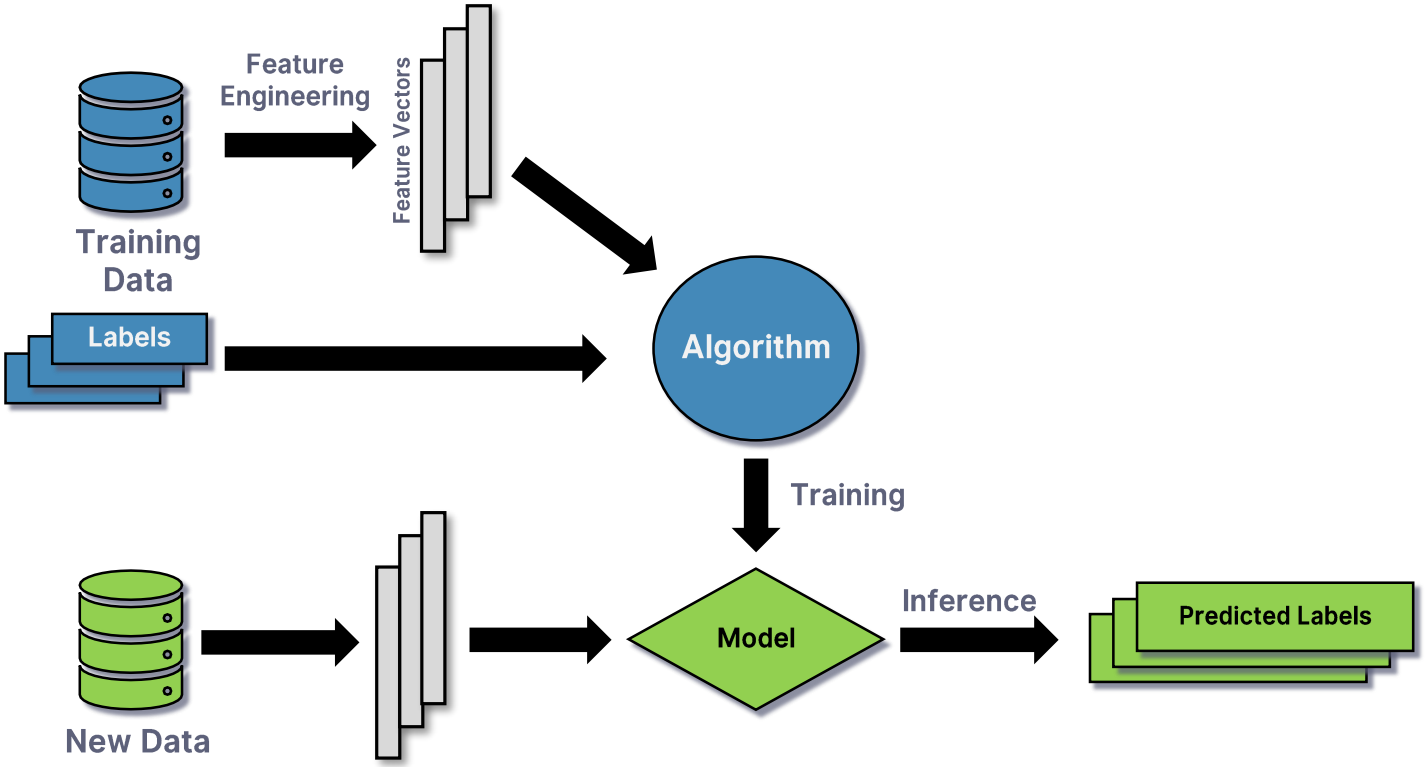
Deep Learning

Learning **representations** with **deep** neural networks

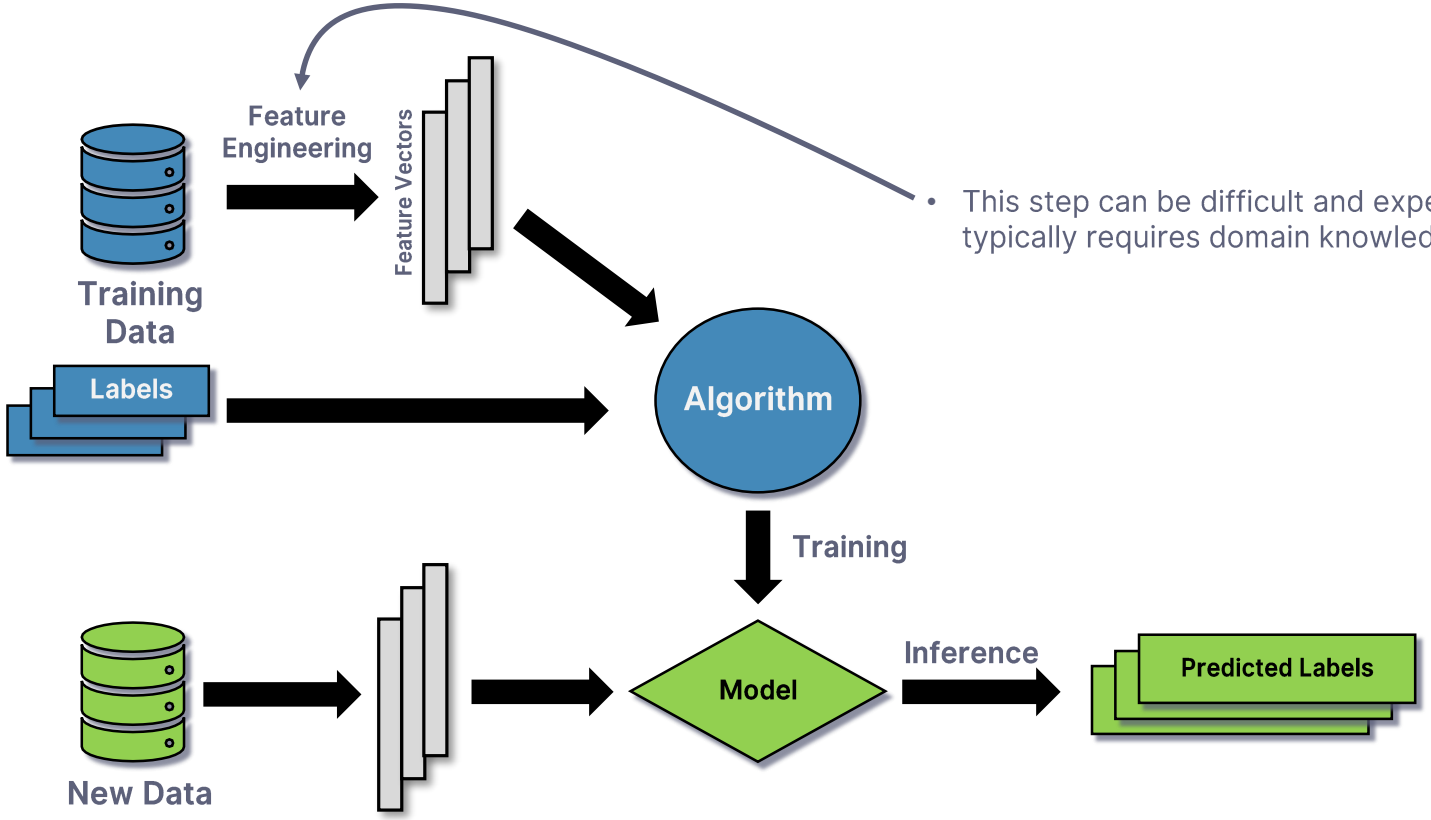
Faculty of Mathematics and Computer Science, University of Bucharest
and
Sparktech Software

Academic Year 2018/2019, 1st Semester

Typical "Shallow" Learning Flow

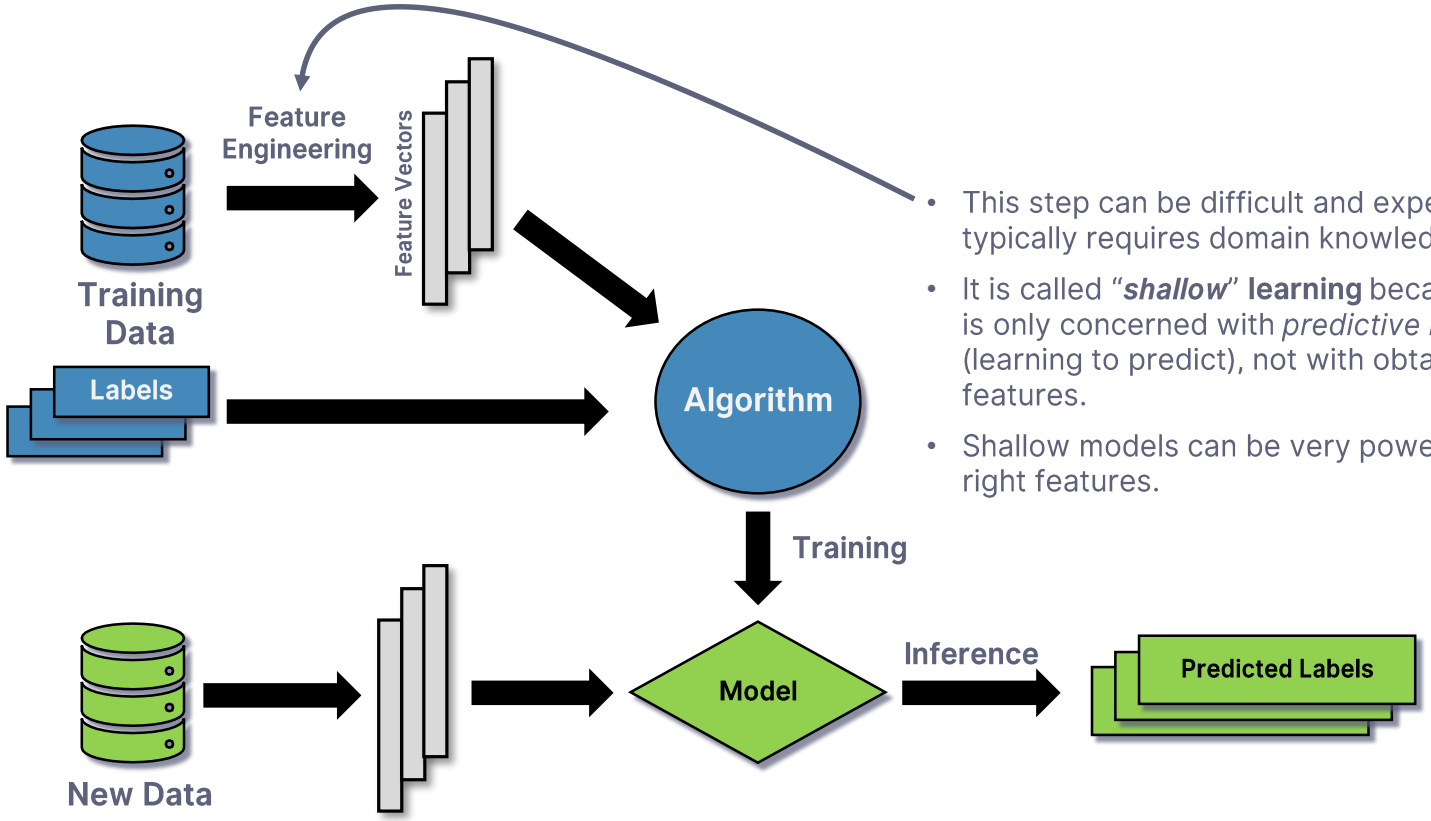


Typical "Shallow" Learning Flow



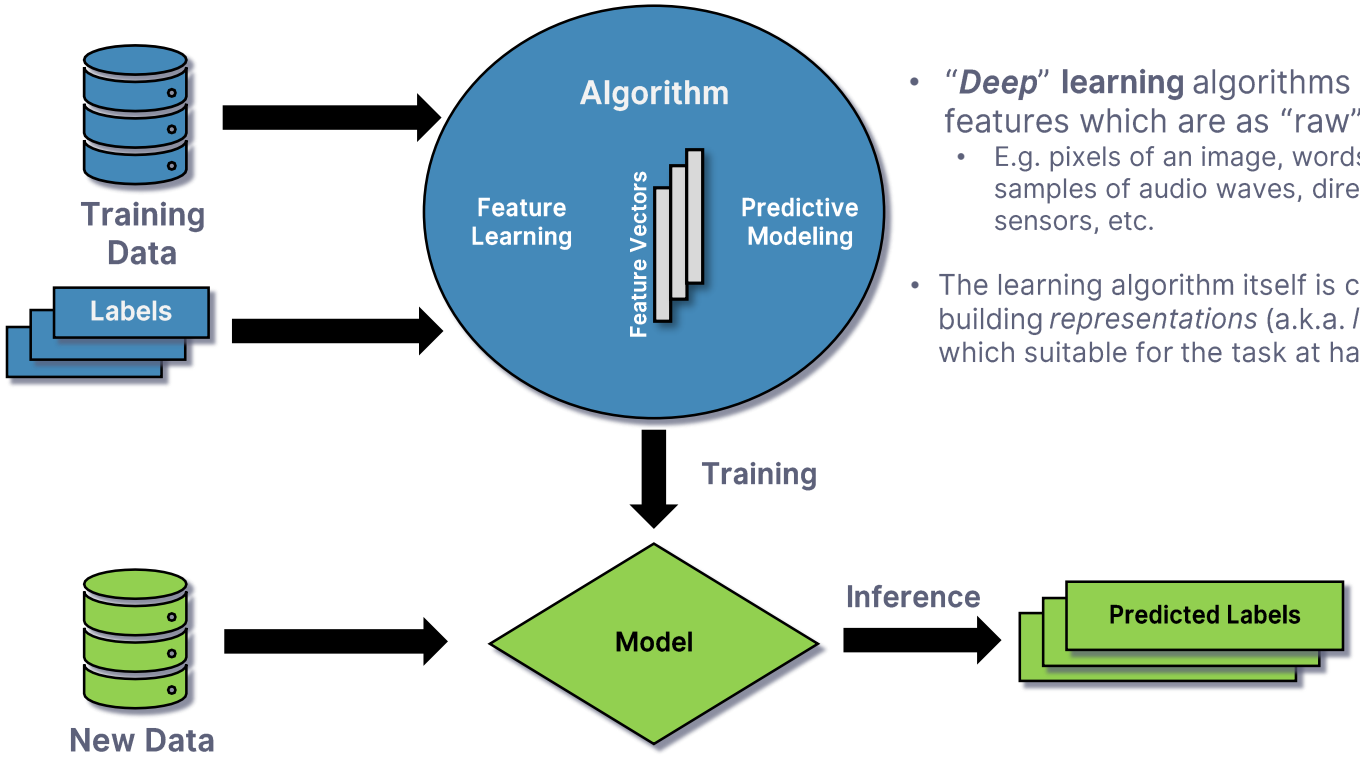
- This step can be difficult and expensive and typically requires domain knowledge.

Typical "Shallow" Learning Flow



- This step can be difficult and expensive and typically requires domain knowledge.
- It is called "***shallow***" **learning** because the algorithm is only concerned with *predictive modeling* (learning to predict), not with obtaining good features.
- Shallow models can be very powerful, given the right features.

Typical "Deep" Learning Flow



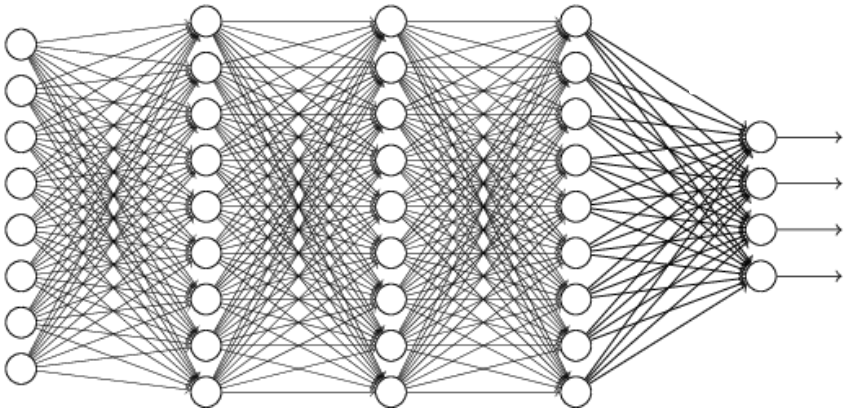
- "***Deep***" **learning** algorithms usually receive features which are as "raw" as possible:
  - E.g. pixels of an image, words or characters of text, samples of audio waves, direct measurements of sensors, etc.
- The learning algorithm itself is concerned with building *representations* (a.k.a. *learning features*) which suitable for the task at hand.

Deep Learning vs. Shallow Learning

- "**Shallow**" **learning** algorithms are more concerned with *predictive modelling*.
  - This usually requires *prior domain knowledge*.
  - Most effort on the developer's part is spent on *feature engineering* and *feature selection*.
- **Deep learning** algorithms are concerned with both learning useful *representations* for inputs and *predictive modelling*.
  - Inputs are supplied in fairy "*raw*" form (e.g. pixels of an image) and this requires *little to no prior domain knowledge*.
  - Most developer effort is spent on *optimizing hyperparameters*.
  - Deep models have considerable more parameters to learn, so they require *much more training data* than shallow models and are very *computationally intensive*.
- There is usually a tradeoff between *feature engineering* and model *"depth"*.

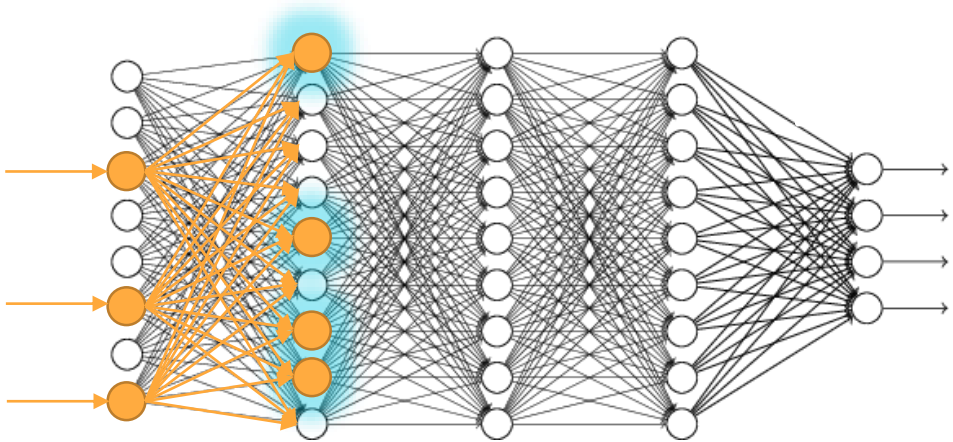
Deep Neural Networks

- A neural network is considered ”**deep**” if it has more than one hidden layer.



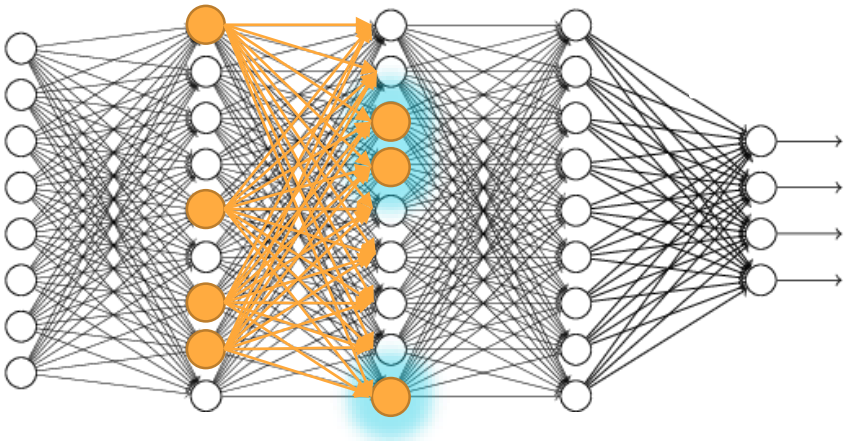
Deep Neural Networks

- A neural network is considered "**deep**" if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.



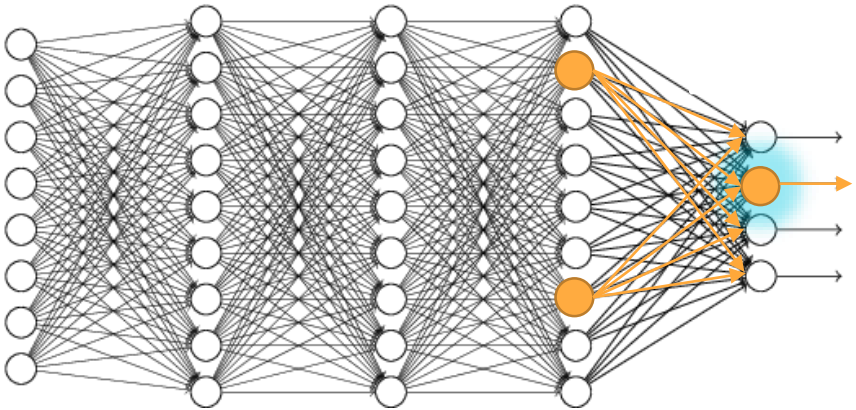
Deep Neural Networks

- A neural network is considered "**deep**" if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.
  - The first layer causes another *activation pattern* in the second layer and so on.



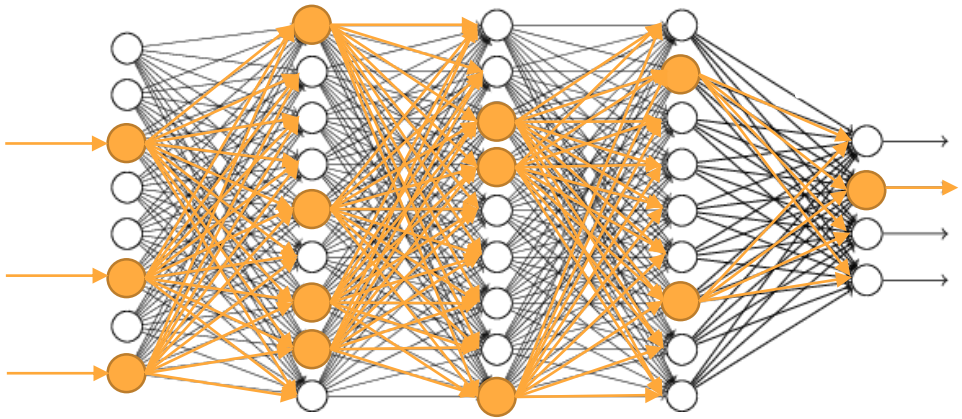
Deep Neural Networks

- A neural network is considered ”**deep**” if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.
  - The first layer causes another *activation pattern* in the second layer and so on.
  - The last layer makes a *prediction*, based the on *activation pattern* of the layer behind it.



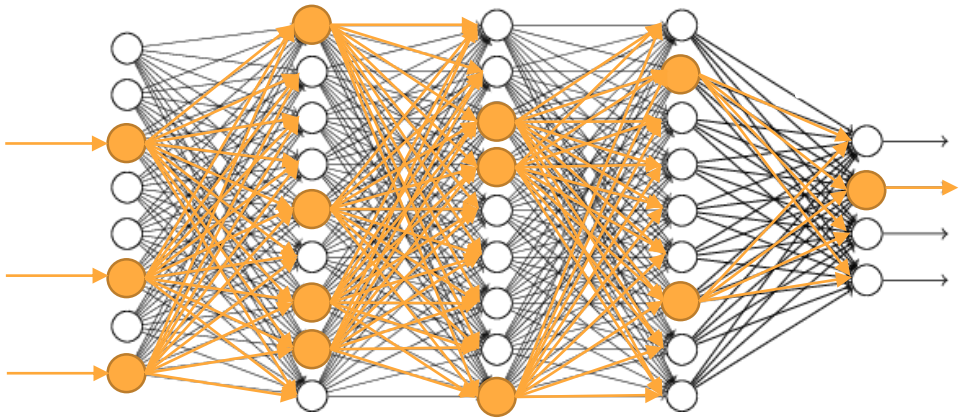
Deep Neural Networks

- A neural network is considered ”**deep**” if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.
  - The first layer causes another *activation pattern* in the second layer and so on.
  - The last layer makes a *prediction*, based the on *activation pattern* of the layer behind it
- In other words, with each layer, the network **gradually builds more and more abstract representations** of the input and the final layer acts like a **predictive model**.

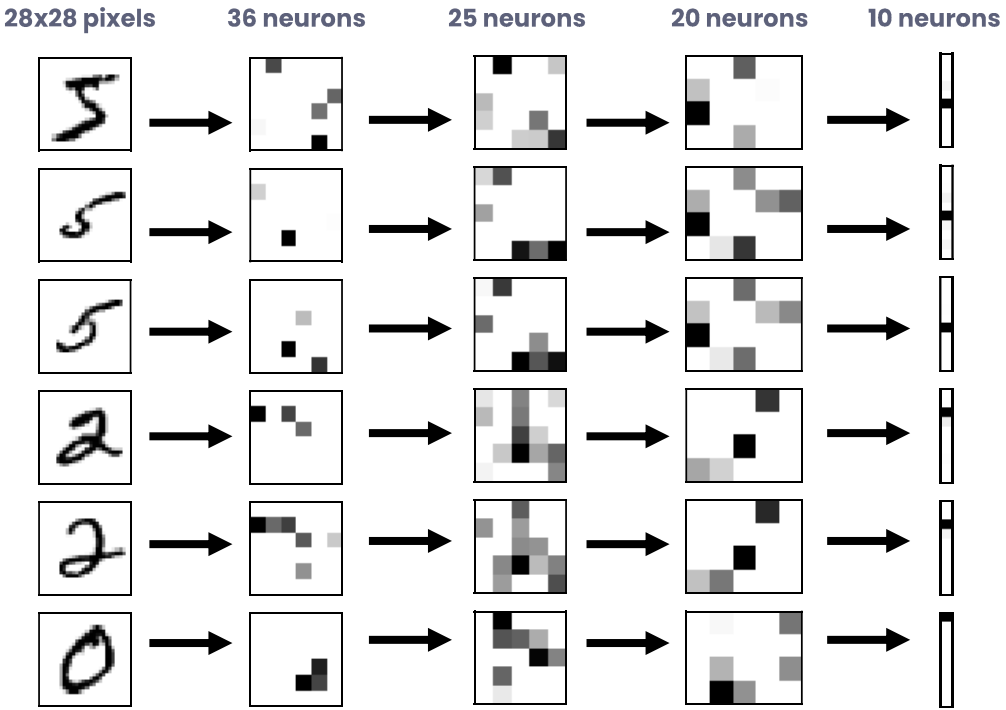


Deep Neural Networks

- A neural network is considered ”**deep**” if it has more than one hidden layer.
  - An input causes an *neuron activation pattern* in the first hidden layer.
  - The first layer causes another *activation pattern* in the second layer and so on.
  - The last layer makes a *prediction*, based the on *activation pattern* of the layer behind it
- In other words, with each layer, the network **gradually builds more and more abstract representations** of the input and the final layer acts like a **predictive model**.
- The *layered structure* of a **deep neural network** makes it the perfect candidate for a **deep learning algorithm**.

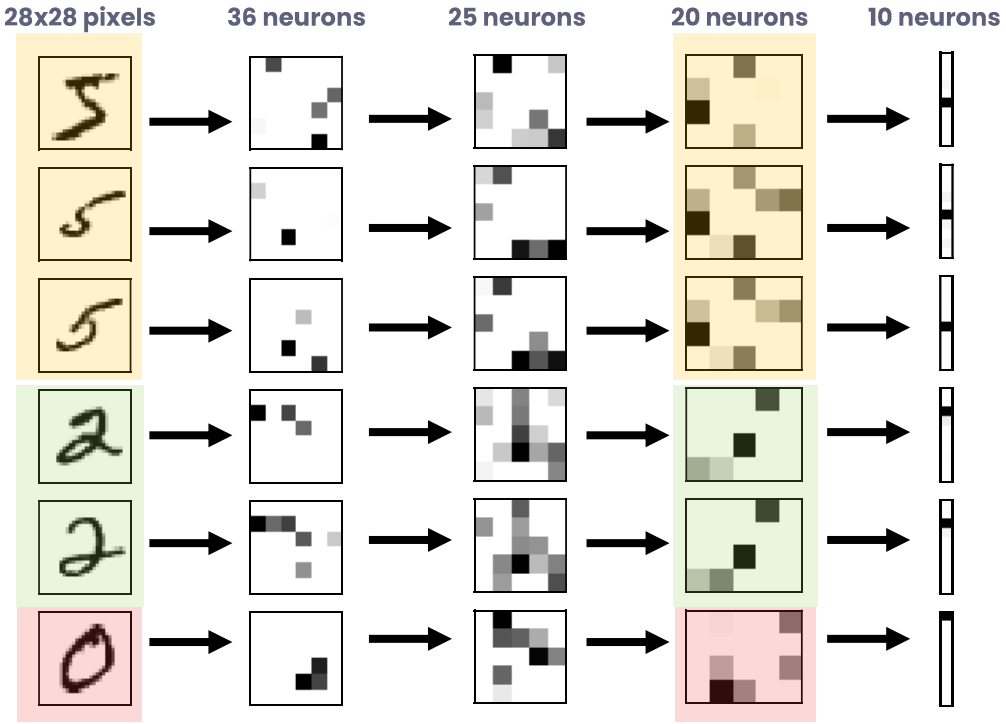


Deep Neural Networks



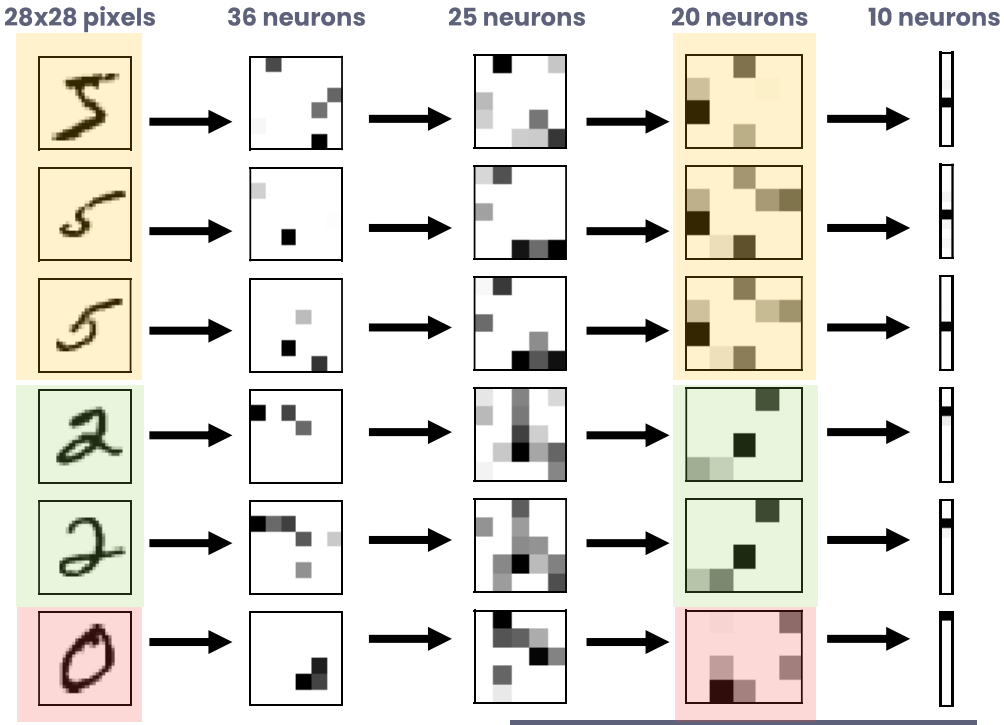
- Every layer produces an *abstract representation* of the input.

Deep Neural Networks



- Every layer produces an *abstract representation* of the input.
- With every layer, the representation is gradually more similar between examples of the same class.

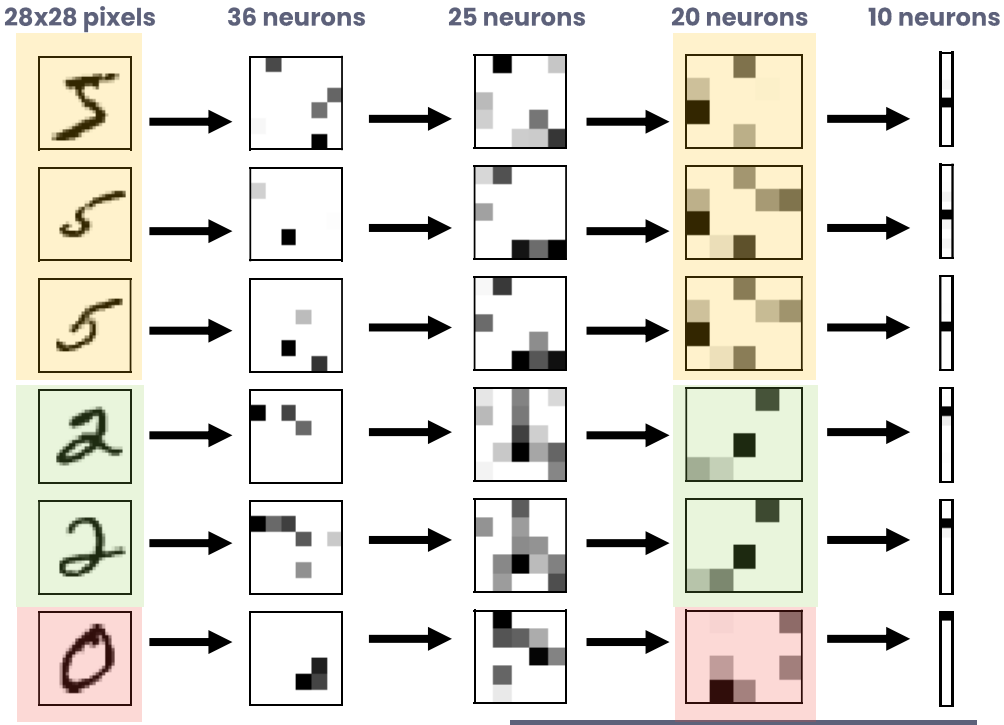
Deep Neural Networks



On the last hidden layer representations have to be linearly separable.

- Every layer produces an *abstract representation* of the input.
- With every layer, the representation is gradually more similar between examples of the same class.
- This makes the *prediction* job easy for the last layer.

Deep Neural Networks



On the last hidden layer representations have to be linearly separable.

- Every layer produces an *abstract representation* of the input.
- With every layer, the representation is gradually more similar between examples of the same class.
- This makes the *prediction* job easy for the last layer.
- Every layer is "**fully connected**" to the one after it.

  $\Rightarrow$ 29,915 parameters to learn! (and this is a relatively small network)

The Trouble with Deep Neural Networks

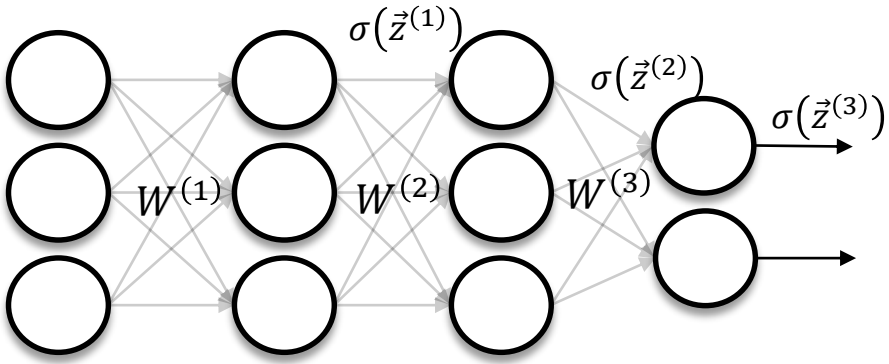
- Training and using deep neural networks in practice was *infeasible* for many years.
- Learning was extremely slow.
  - A network with a few hidden layers can reach *millions of parameters*.
  - Training is very *computationally intensive*.
  - This also means that researchers had to wait a long time to see if a new idea worked.
- Deep networks are very *complex models* and are prone to *overfitting*
  - To avoid this, they require *considerably more training data* than shallow methods (sometimes millions of training samples).
- The network structure itself (i.e. *number of layers* and number of neurons per layer) is a very hard *hyperparameter* to tune, since there are lots of possibilities.

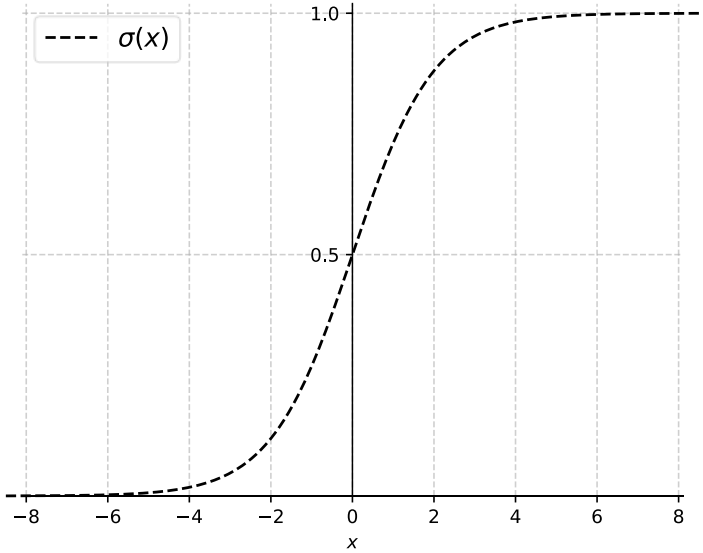
What has changed?

- In the late 2000s and early 2010s, *Deep Neural Networks* and *Deep Learning* have seen a major increase in popularity and performance.
- Aside from the major increase in **data availability** and **computing power**, an important reason for the recent breakthrough in *Deep Learning* are the advances made by researchers on the **algorithmic** side.
  - Better **activation** functions (e.g. ReLU)
  - Better **optimization** algorithms (e.g. Momentum, RMSProp, Adam)
  - Better network layer **structure** (e.g. Convolutions, LSTMs, Residual Networks).
  - Better **regularization** techniques (e.g. Dropout, BatchNorm).
  - Deep Learning **frameworks** (e.g. TensorFlow, Theano, PyTorch, Caffe).

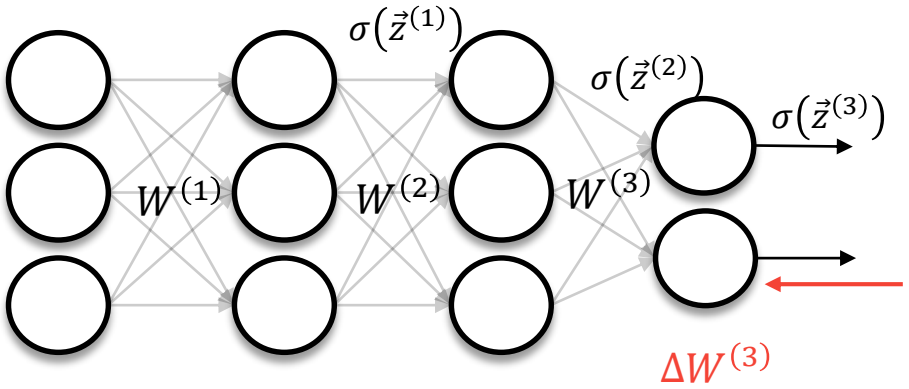
Better
Activation Functions

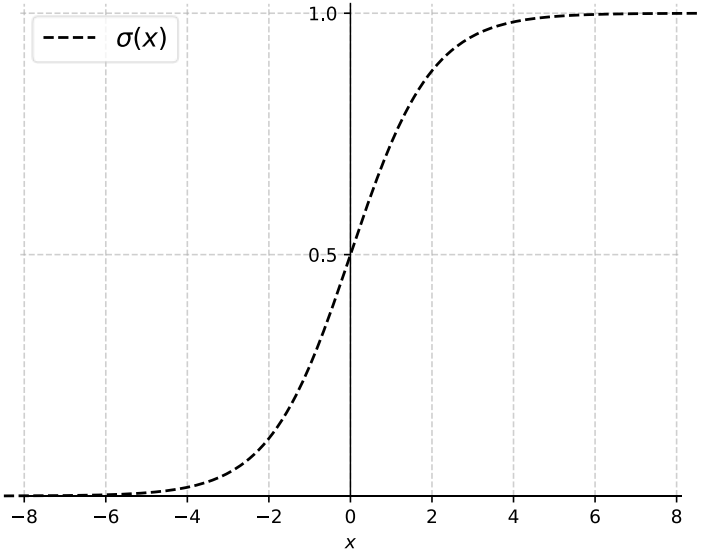
The Trouble with Sigmoids





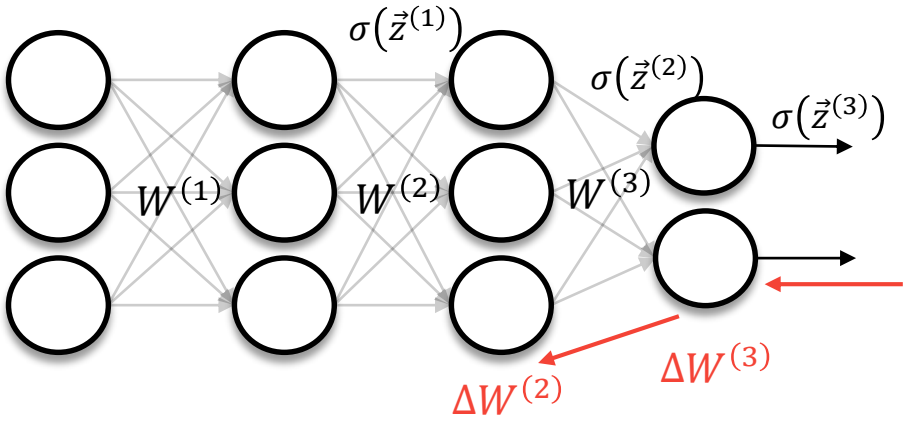
The Trouble with Sigmoids

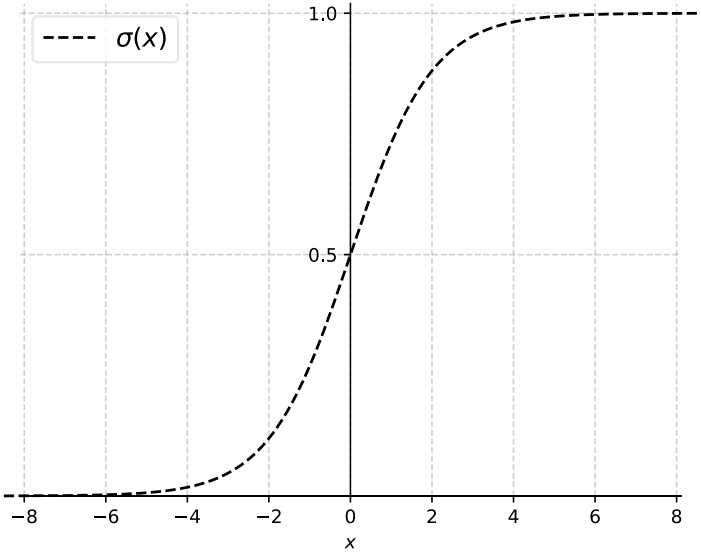




$$\Delta W^{(3)} = (\ldots) \cdot \sigma'(\vec{z}^{(3)})$$

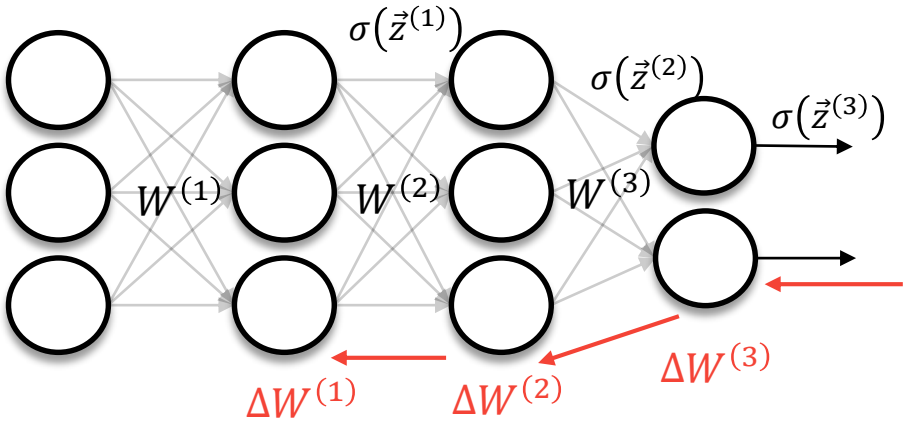
The Trouble with Sigmoids

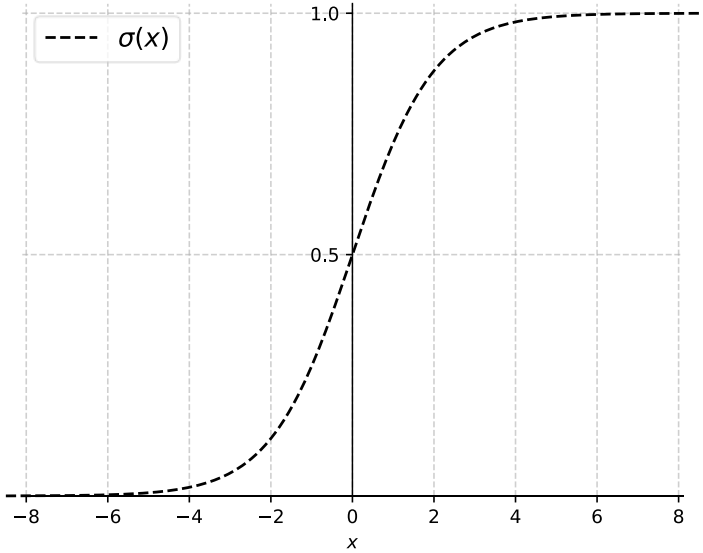




$$\Delta W^{(3)} = (\ldots) \cdot \sigma'(\vec{z}^{(3)})$$
$$\Delta W^{(2)} = (\ldots) \cdot \sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})$$

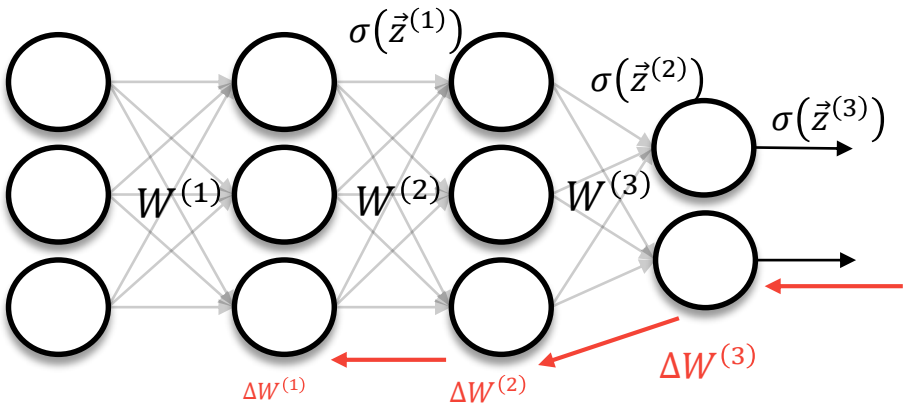
The Trouble with Sigmoids

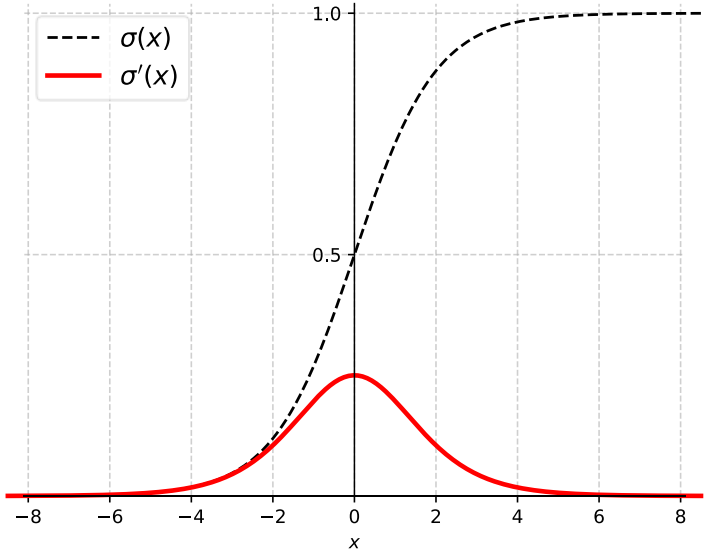




$$\Delta W^{(3)} = (\dots) \cdot \sigma'(\vec{z}^{(3)})$$
$$\Delta W^{(2)} = (\dots) \cdot \sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})$$
$$\Delta W^{(1)} = (\dots) \cdot \sigma'(\vec{z}^{(1)})\sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})$$

The Trouble with Sigmoids





$$\Delta W^{(3)} = (\ldots) \cdot \sigma'(\vec{z}^{(3)})$$

$$\Delta W^{(2)} = (\ldots) \cdot \sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})$$

$$\Delta W^{(1)} = (\ldots) \cdot \underbrace{\sigma'(\vec{z}^{(1)})\sigma'(\vec{z}^{(2)})\sigma'(\vec{z}^{(3)})}_{\text{Very small number.}}$$

$\sigma'(x)$ has a maximum value of 0.25 for $x = 0$ and it drops very quickly (e.g. $\sigma'(10) \approx 0.00004$).

The Trouble with Sigmoids

- The derivative of the *logistic function* $\sigma'(x)$ has a maximum value of 0.25 for $x = 0$ and it drops very quickly (e.g. $\sigma'(10) \approx 0.00004$).
- Because of the *chain rule*, the weight updates in one layer are multiplied by the gradients of all layers after it.
  - This causes updates in early layers to be very small and learning to be very slow.
  - The updates can even be 0 because of *floating point precision*.
  - A neuron with an output very close to 0 or 1 and, thus, a very small update, is said to be "*saturated*"
  - The problem gets worse for *deeper networks*.
- This is known as the **vanishing gradient** problem.
- Another issue with the logistic (and other sigmoids) is that it is *computationally expensive*, since it requires computing the exponential e^{-x}.

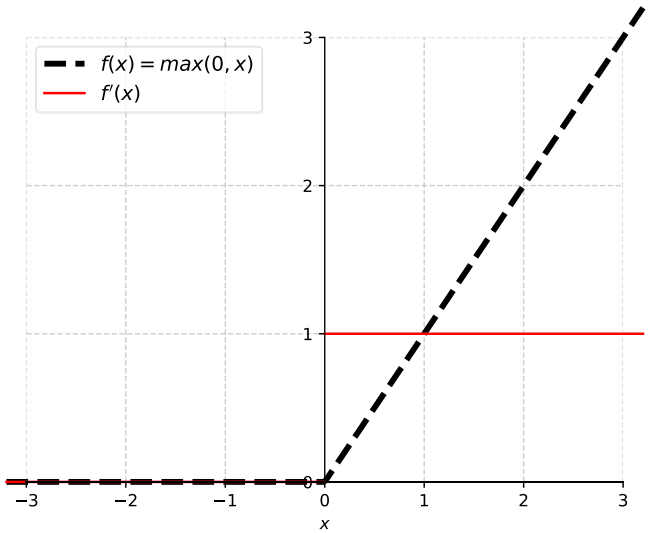
Rectified Linear Units

- The linear activation $f(x) = x$ does not suffer from *vanishing gradient problem*, but it also does not benefit from multiple layers.

Rectified Linear Units

- The linear activation $f(x) = x$ does not suffer from *vanishing gradient problem*, but it also does not benefit from multiple layers.
- The **Rectified Linear Unit** (**ReLU**) is a *non-linear activation* function which takes only the positive part of the linear activation.

$$f(x) = \max(0, x) \qquad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$



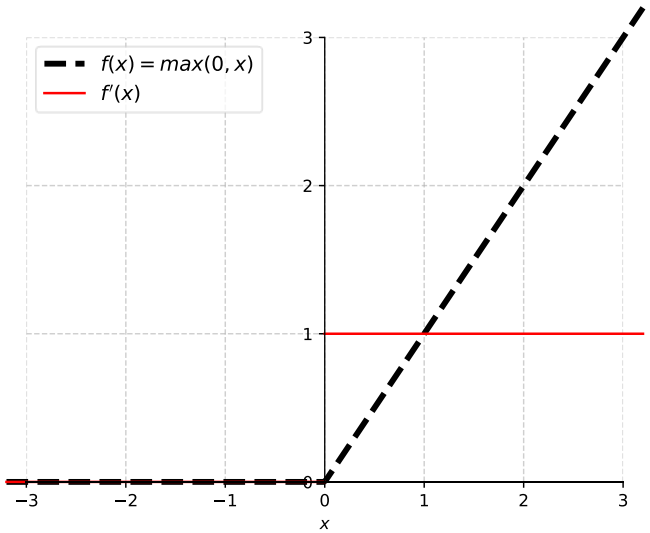
f is not differentiable in 0, but this not a problem in practice

Rectified Linear Units

- The linear activation $f(x) = x$ does not suffer from *vanishing gradient problem*, but it also does not benefit from multiple layers.
- The **Rectified Linear Unit** (**ReLU**) is a *non-linear activation* function which takes only the positive part of the linear activation.

$$f(x) = \max(0, x) \qquad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

- *ReLU* neurons cannot *saturate* on the positive side so they don't suffer from the *vanishing gradient* problem
- The *ReLU* activation is much easier to compute.
- *ReLUs* output 0 in the negative range which encourages activation sparsity.



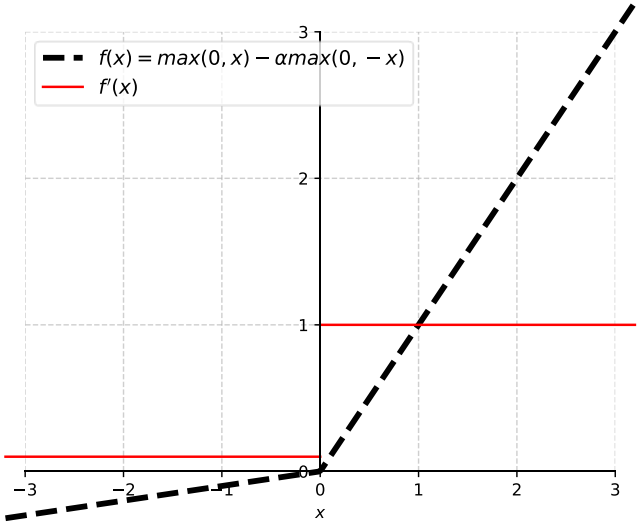
f is not differentiable in 0, but this not a problem in practice

Dying ReLU Problem

- Because the output of a ReLU neuron is constant in the negative range, the slope is 0.
- This means that a *ReLU* neuron can ***"die".***
  - i.e. get stuck on the negative side with the output always 0 and very small chances of recovering.
- This effect can be good to some extent, since it encourages activation *sparsity*, but if it happens too much it can render a very large part of the network *essentially useless*.
- It can be mitigated by setting a *smaller learning rate* or by using a *ReLU* variant (e.g. *LeakyReLU*, *PreELU*)

ReLU Variants

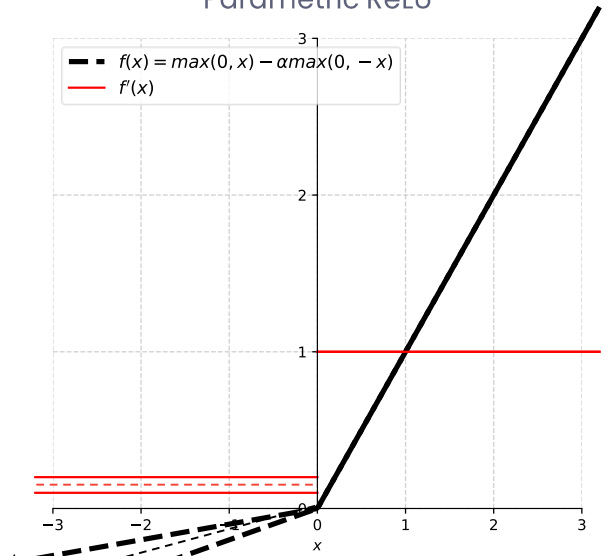
Leaky ReLU



$\alpha = 0.1$

α is a **hyperparameter**.
(It is set before training, and remains constant afterwards).

PReLU
Parametric ReLU



α is a **parameter**.
(It is *learned* together with other parameters of the network).

Better Optimizers

Stochastic, Batch and Mini-Batch

- **“Stochastic” Gradient Descent** is used for the case in which the algorithm computes the gradient and makes *one weight update for one training sample* at a time.
    - The can lead slow convergence because of *zig-zagging* of the weight vector instead of always taking steps towards the minimum.

Stochastic, Batch and Mini-Batch

- **“Stochastic” Gradient Descent** is used for the case in which the algorithm computes the gradient and makes *one weight update for one training sample* at a time.
  - The can lead slow convergence because of *zig-zagging* of the weight vector instead of always taking steps towards the minimum.
- Ideally, we want to make a *single weight update for the average gradient of all training samples*.
  - This is known as **Batch Gradient Descent**, or sometimes **“Vanilla” Gradient Descent**.
  - This method computes the *optimal direction for every update*, but every step involves *lots of computations*.

Stochastic, Batch and Mini-Batch

- **"Stochastic" Gradient Descent** is used for the case in which the algorithm computes the gradient and makes *one weight update for one training sample* at a time.
  - The can lead slow convergence because of *zig-zagging* of the weight vector instead of always taking steps towards the minimum.
- Ideally, we want to make a *single weight update for the average gradient of all training samples*.
  - This is known as **Batch Gradient Descent**, or sometimes **"Vanilla" Gradient Descent**.
  - This method computes the *optimal direction for every update*, but every step involves *lots of computations*.
- The most common method is to make *one weight update based on the average gradient for a (small) subset of data*.
  - This is called **Mini-Batch Gradient Descent**.
  - It usually *converges faster* the *Batch*, even if it takes more steps, because every step is *computed much faster*.

Mini-Batch Gradient Descent is almost always used in practice, but sometimes it is referred to as **SGD**.

Problems with Gradient Descent

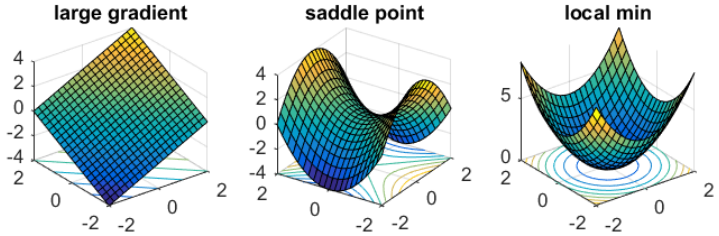
- Even the optimal, but slow, **Batch Gradient Descent** does not guarantee *good convergence*.

Problems with Gradient Descent

- Even the optimal, but slow, **Batch Gradient Descent** does not guarantee *good convergence*.
- The *error surface* of a *deep neural network* is **non-convex** which means that GD can easily get stuck in **local minima**.

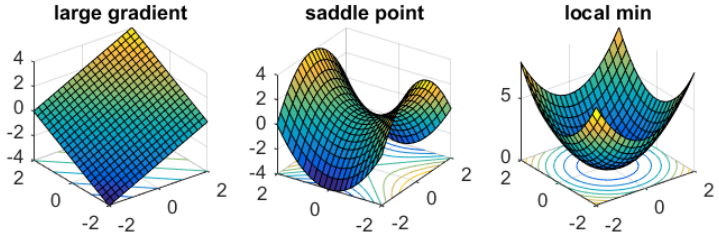
Problems with Gradient Descent

- Even the optimal, but slow, **Batch Gradient Descent** does not guarantee *good convergence*.
- The *error surface* of a *deep neural network* is **non-convex** which means that GD can easily get stuck in **local minima**.
- However, it turns out that **saddle points**, not local minima, are a worse problem for GD convergence.
  - *Saddle points* are locations where some dimensions slope up and others slope down, which makes the gradient ~0 in all directions.



Problems with Gradient Descent

- Even the optimal, but slow, **Batch Gradient Descent** does not guarantee *good convergence*.
- The *error surface* of a *deep neural network* is **non-convex** which means that GD can easily get stuck in **local minima**.
- However, it turns out that **saddle points**, not local minima, are a worse problem for GD convergence.
  - *Saddle points* are locations where some dimensions slope up and others slope down, which makes the gradient ~0 in all directions.



- Some of these issues can be alleviated by choosing a proper **learning rate**, but this is not trivial.
  - Small *learning rate* means slow convergence, *large learning* rate means fluctuations around the minimum or divergence.
  - Another issue is that the *same learning rate* applies to all weights. We might want more active neurons to have a smaller learning rate than neurons which are rarely active.

Exponential Moving Average

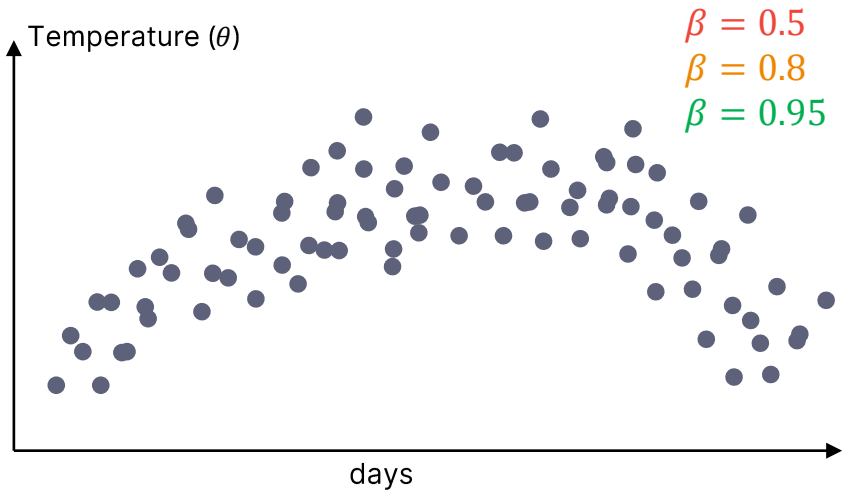
- A **moving average** is a calculation used to smooth out random fluctuations in a (time-)series of data points.

Exponential Moving Average

- A **moving average** is a calculation used to smooth out random fluctuations in a (time-)series of data points.
- In an **exponential moving average** (or **exponentially weighted moving average**), more recent points have exponentially more impact on the average.

$$v_0 = 0$$
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

v_t approximates the average over $\frac{1}{1-\beta}$ values

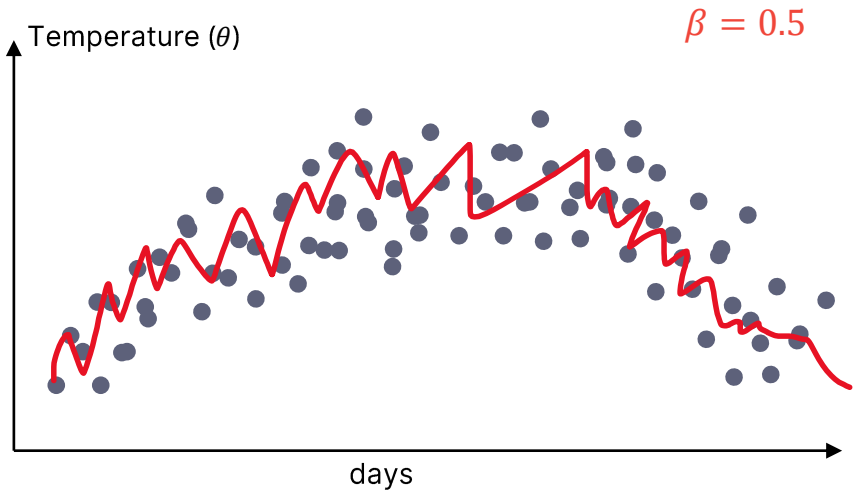


Exponential Moving Average

- A **moving average** is a calculation used to smooth out random fluctuations in a (time-)series of data points.
- In an **exponential moving average** (or **exponentially weighted moving average**), more recent points have exponentially more impact on the average.

$$v_0 = 0$$
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

v_t approximates the average over $\frac{1}{1-\beta}$ values

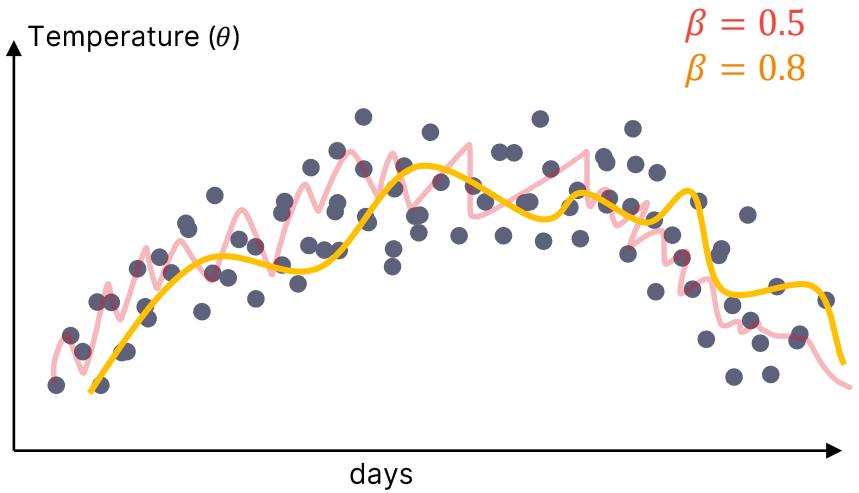


Exponential Moving Average

- A **moving average** is a calculation used to smooth out random fluctuations in a (time-)series of data points.
- In an **exponential moving average** (or **exponentially weighted moving average**), more recent points have exponentially more impact on the average.

$$v_0 = 0$$
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

v_t approximates the average over $\frac{1}{1-\beta}$ values

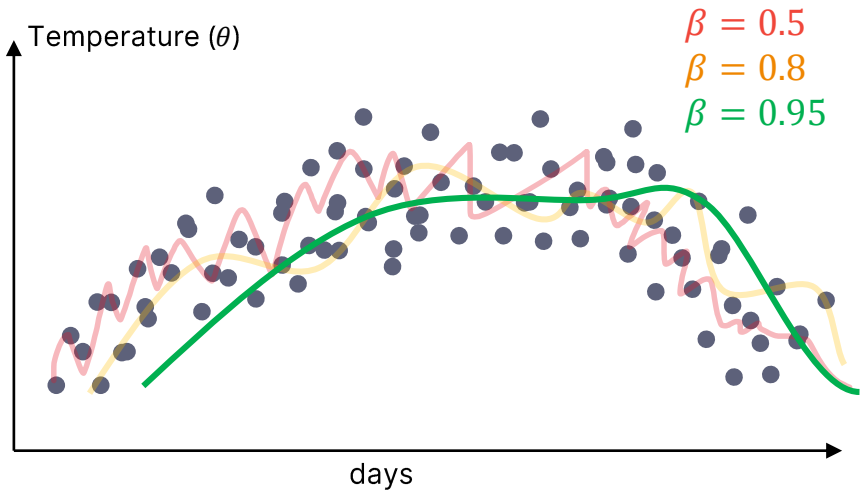


Exponential Moving Average

- A **moving average** is a calculation used to smooth out random fluctuations in a (time-)series of data points.
- In an **exponential moving average** (or **exponentially weighted moving average**), more recent points have exponentially more impact on the average.

$$v_0 = 0$$
$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

v_t approximates the average over $\frac{1}{1-\beta}$ values



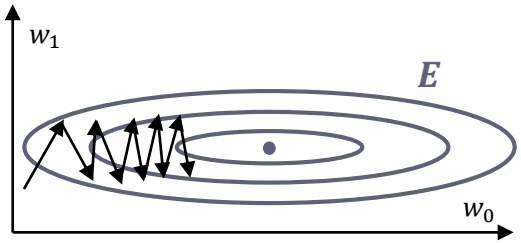
Momentum

- One case in which SGD is slow is when the error surface is more steep in one direction then in another.

$$\Delta w^{t+1} = -\eta \frac{\partial E}{\partial w^t}$$

w^t is the weight at time t.

  - This causes SGD to oscillate from one side to the other, while making little progress towards the minimum.



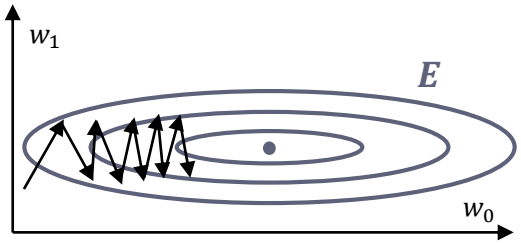
Momentum

- One case in which SGD is slow is when the error surface is more steep in one direction then in another.

$$\Delta w^{t+1} = -\eta \frac{\partial E}{\partial w^t}$$

w^t is the weight at time t.

  - This causes SGD to oscillate from one side to the other, while making little progress towards the minimum.



- **Momentum** is a method which accelerates SGD in a certain direction if several consecutive updates push it towards it.

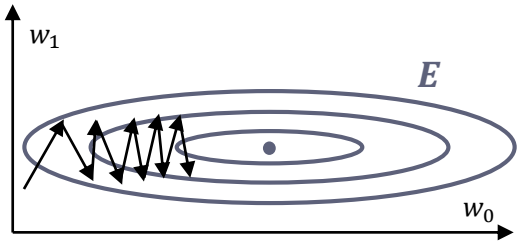
Momentum

- One case in which SGD is slow is when the error surface is more steep in one direction then in another.

$$\Delta w^{t+1} = -\eta \frac{\partial E}{\partial w^t}$$

w^t is the weight at time t.

  - This causes SGD to oscillate from one side to the other, while making little progress towards the minimum.

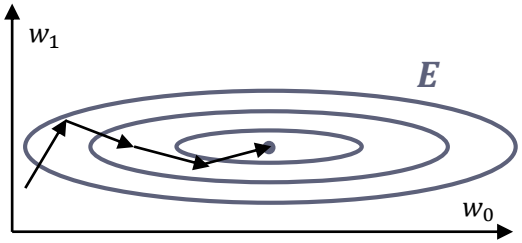


- **Momentum** is a method which accelerates SGD in a certain direction if several consecutive updates push it towards it.
- It works by keeping an *exponential moving average of gradients* (the "**velocity**") and using it to update the weights, instead of using the gradient directly.

$$v^t = \gamma v^{t-1} + (1-\gamma)\frac{\partial E}{\partial w^t} \qquad \Delta w^{t+1} = -\eta \Delta v^t$$

γ is usually ~0.9

  - The velocities in relevant directions grow over time.
  - The velocities in oscillating directions cancel out.



- Momentum can also help to avoid getting stuck in certain local minima.

Adaptive Learning Rate

- **Adagrad**, **Adadelta** and **RMSProp** are optimization techniques which use an **adaptive learning rate** (i.e. the learning rate adapts to each weight at each step).

Adaptive Learning Rate

- **Adagrad**, **Adadelta** and **RMSProp** are optimization techniques which use an **adaptive learning rate** (i.e. the learning rate adapts to each weight at each step).
- **RMSProp** (**R**oot **M**ean **S**quare **Prop**) tries to make learning rate smaller for weights which get updated frequently. It does this by keeping a *moving average of squared gradients* for each weight and scales the corresponding learning rate by the *root* of this value.

$$\text{MeanSquare}(w^t) = \gamma \text{MeanSquare}(w^{t-1}) + (1-\gamma)\left(\frac{\partial E}{\partial w^t}\right)^2$$

Exponential moving average of squared gradients.

$$\Delta w^{t+1} = -\frac{\eta}{\sqrt{\text{MeanSquare}(w^t)}}\frac{\partial E}{\partial w^t}$$

Adaptive Learning Rate

- **Adagrad**, **Adadelta** and **RMSProp** are optimization techniques which use an **adaptive learning rate** (i.e. the learning rate adapts to each weight at each step).
- **RMSProp** (**R**oot **M**ean **S**quare **Prop**) tries to make learning rate smaller for weights which get updated frequently. It does this by keeping a *moving average of squared gradients* for each weight and scales the corresponding learning rate by the *root* of this value.

$$\text{MeanSquare}(w^t) = \gamma \text{MeanSquare}(w^{t-1}) + (1-\gamma)\left(\frac{\partial E}{\partial w^t}\right)^2$$

Exponential moving average of squared gradients.

$$\Delta w^{t+1} = -\frac{\eta}{\sqrt{\text{MeanSquare}(w^t)}}\frac{\partial E}{\partial w^t}$$

- **ADAM** (**Ada**ptive **M**oment Estimation) is a combination of *Momentum* and *RMSProp* and it is very common in practice.

$$v^t = \gamma_1 v^{t-1} + (1-\gamma_1)\frac{\partial E}{\partial w^t} \qquad \text{MeanSquare}(w^t) = \gamma_2 \text{MeanSquare}(w^{t-1}) + (1-\gamma_2)\left(\frac{\partial E}{\partial w^t}\right)^2$$

$$\Delta w^{t+1} = -\frac{\eta}{\sqrt{\text{MeanSquare}(w^t)}} v^t$$

In practice, the moving averages have bias-correction.

Keywords

Deep Learning | Shallow Learning | Feature Engineering

Deep Neural Networks

Vanishing Gradient | Rectified Linear Unit | ReLu

Stochastic Gradient Descent | Batch Gradient Descent | Mini-Batch Gradient Descent

Exponential Moving Average | Momentum | RMSProp | ADAM