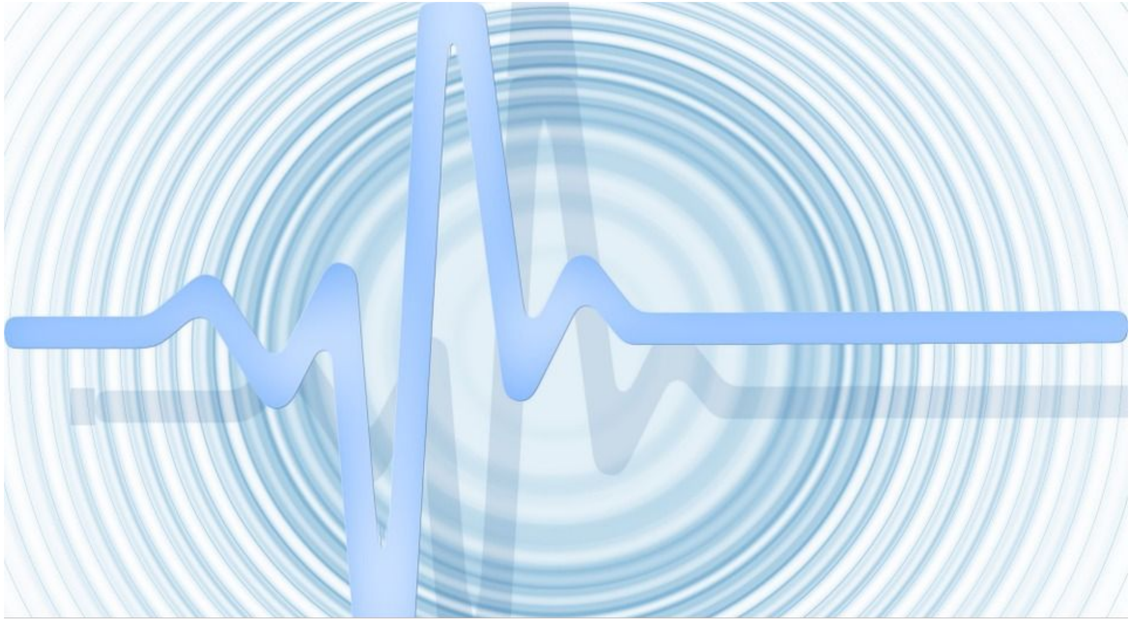
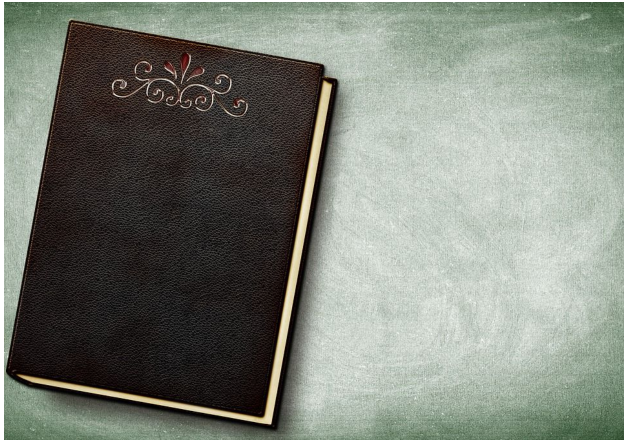


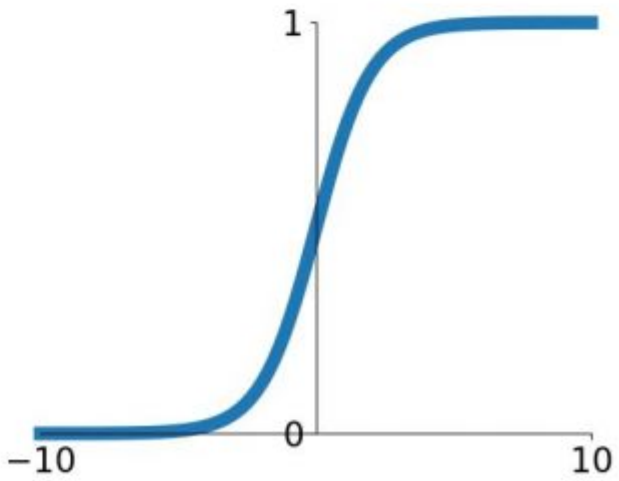
Activation Function

Controls Neuron’s Output



Controls Neuron’s Learning

Sigmoid Function

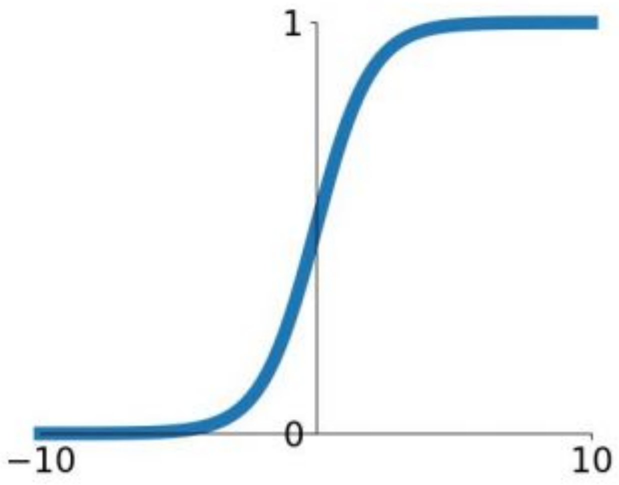


$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

- Squashes output between 0 and 1
- Nice interpretation *i.e* neuron firing or not firing

It has 3 problems.

Sigmoid Function



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

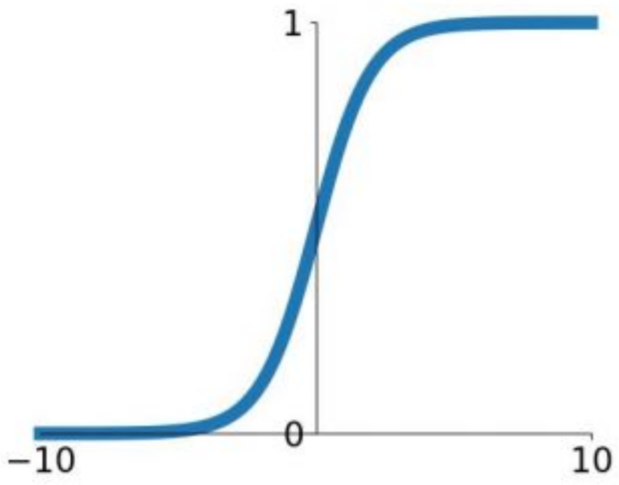
Problem 1

- Vanishing Gradient

Derivative is zero when x> 5 or x <-5

- Weights will not change
- No Learning

Sigmoid Function



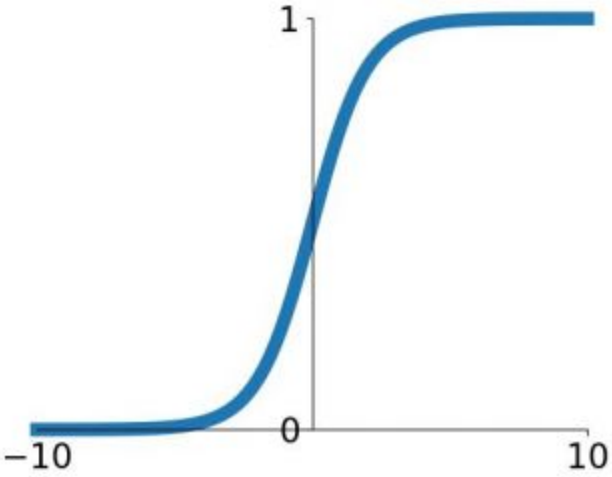
$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Problem 2

- Output is not Zero-centered

Only positive numbers to Next layer

Sigmoid Function

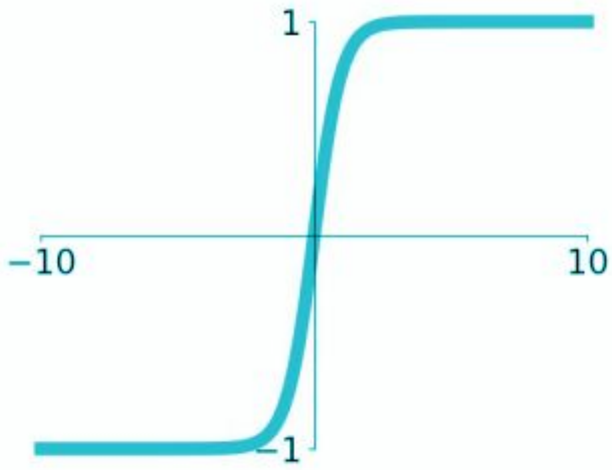


$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Problem 3

- e^y is compute expensive

tanh

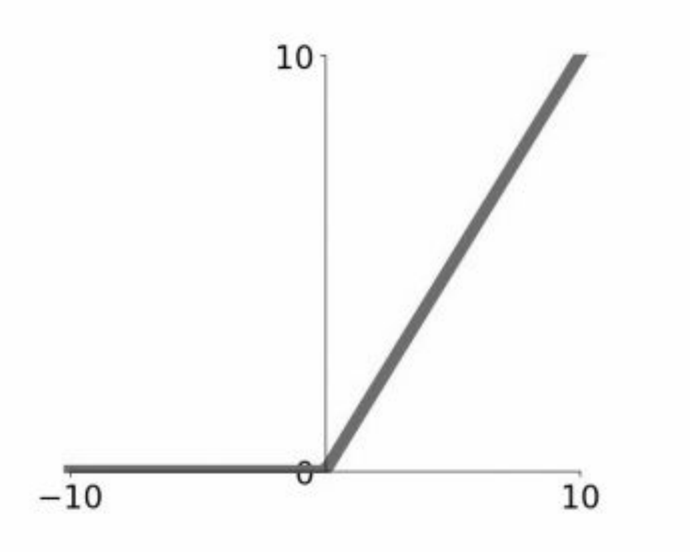


hyperbolic tangent

$$tanh(y) = 2 \cdot \sigma(2y) - 1$$

1. Zero-centered
2. Vanishing gradient
3. Compute expensive

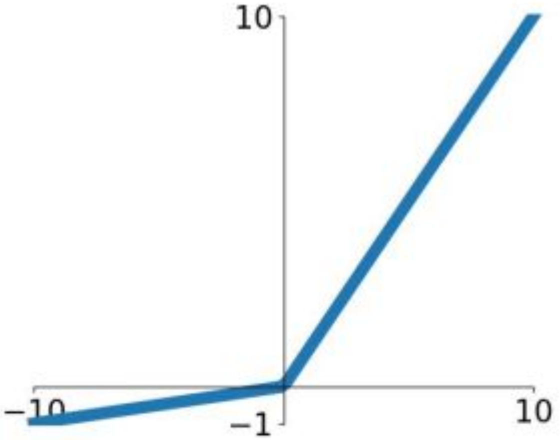
Rectified Linear Unit (ReLU)



$$max(0, x)$$

1. Does not kill gradient (x>0)
2. Compute inexpensive
3. Converges faster
4. No Zero-centered output

Leaky ReLU



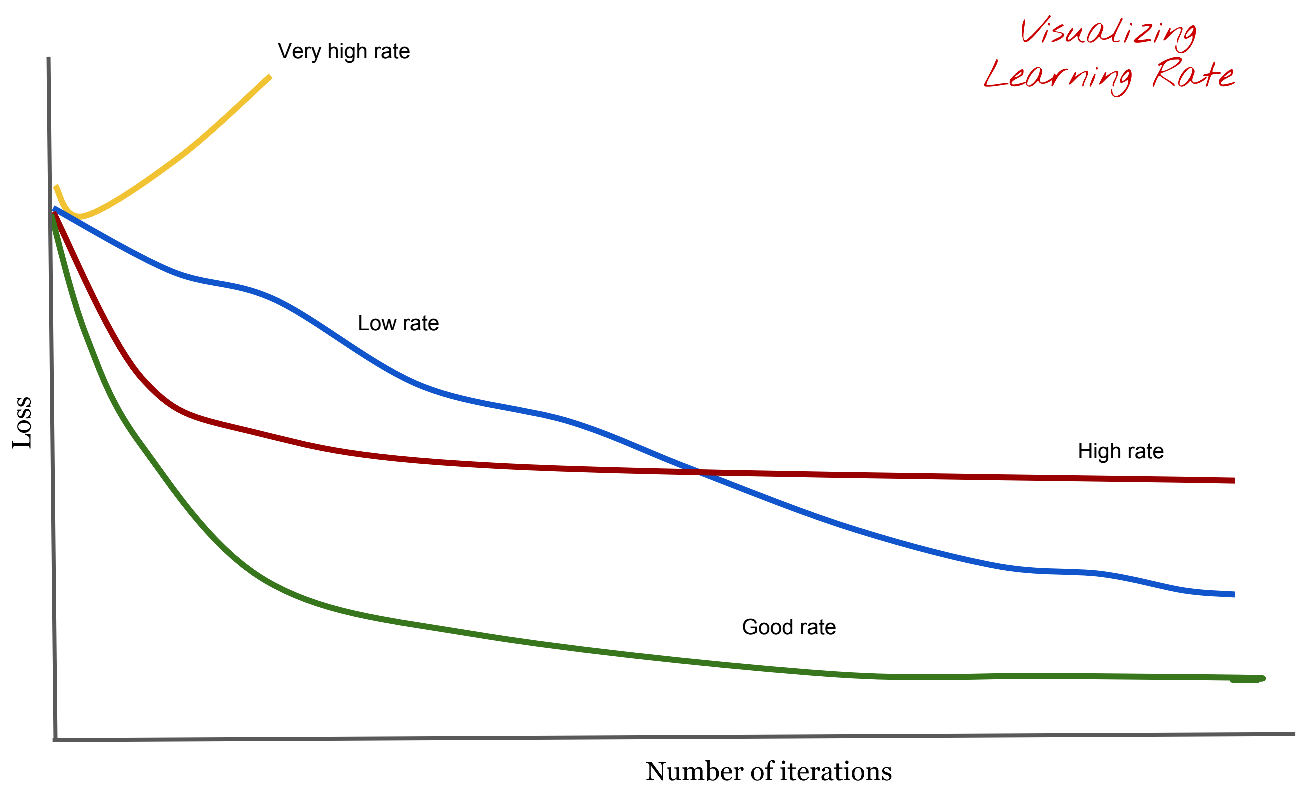
$$max(0.01x, x)$$

1. Does not kill gradient
2. Compute inexpensive
3. Converges faster
4. Somewhat Zero-centered

Which one should we use?

- Use ReLU
- Try out Leaky ReLU
- Try out tanh but don’t expect much
- Minimize use of Sigmoid

Learning Rate


Visualizing Learning Rate
Very high rate
Low rate
High rate
Good rate
Loss
Number of iterations

Learning rate decay

Time based learning rate decay

$$\alpha_t = \frac{\alpha_0}{(1+kt)}$$

$\alpha_0 \rightarrow$ *Initial Learning Rate*

$k \rightarrow$ *Decay rate*

$t \rightarrow$ *Iteration number*

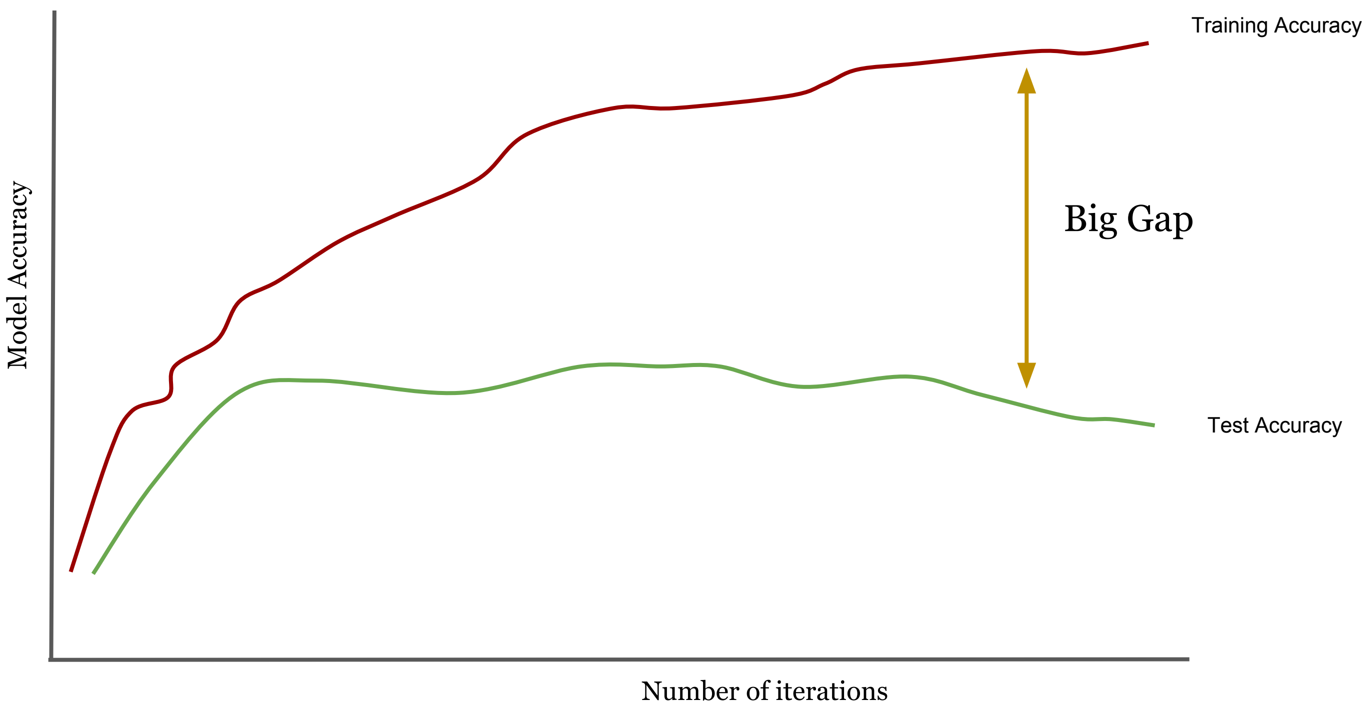
```
sgd_optimizer = tf.keras.optimizers.SGD(lr=0.1, decay=0.001)

model.compile(optimizer=sgd_optimiser, loss='mse’)
```

Learning

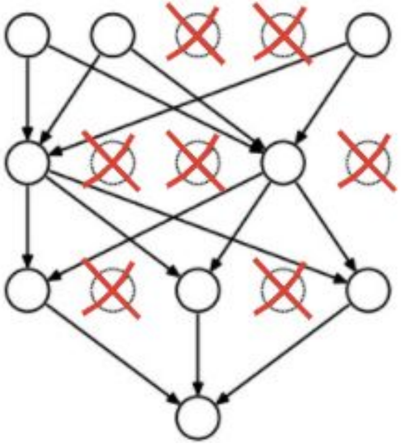
vs

Memorizing

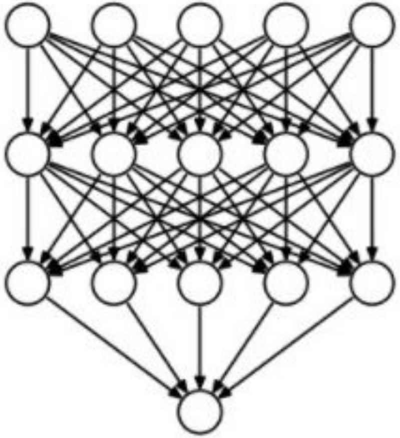
Overfitting



Dropout



TRAINING
Drop=50%



EVALUATION
Keep=100%

Dropout

```
model.add(tf.keras.layers.Dropout(0.4)
```

```
model.add(tf.keras.layers.Dense(200))

model.add(tf.keras.layers.Dropout(0.4))

model.add(tf.keras.layers.Dense(100))
```

Optimizers

$$w_{t+1} = w_t - \alpha \Delta J(w_t)$$

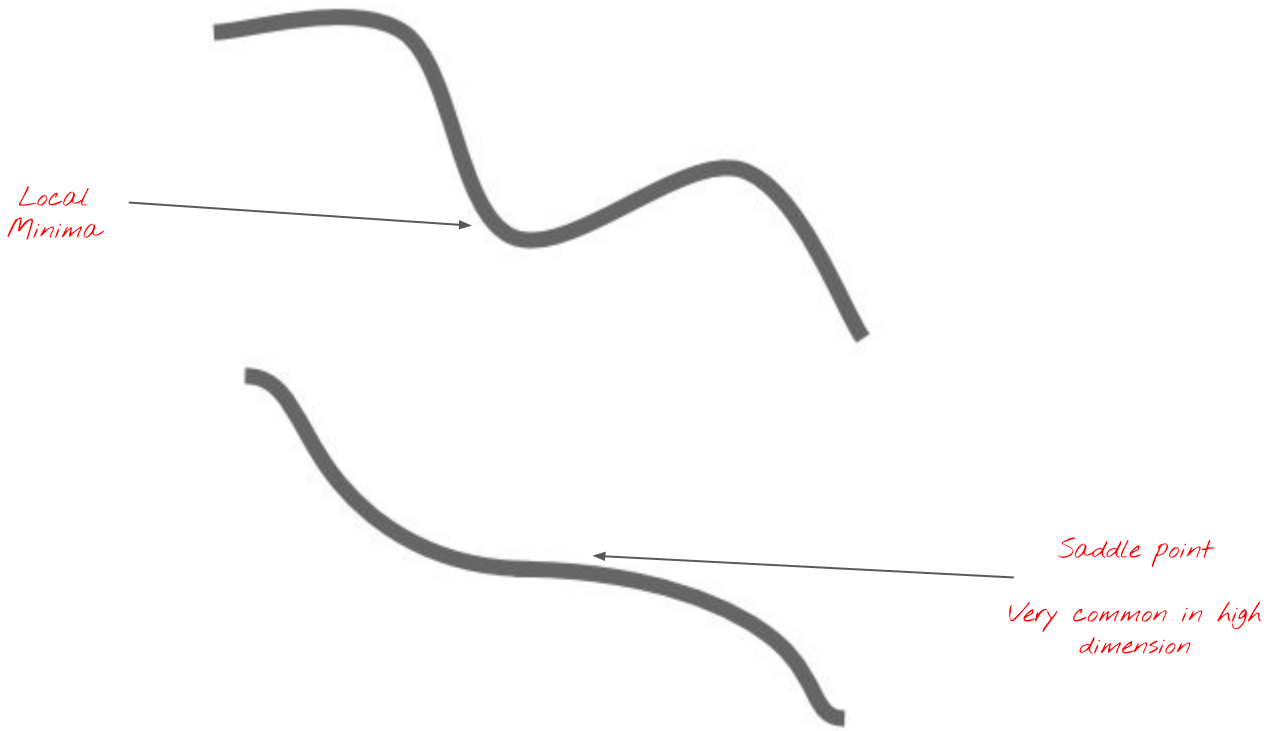
New Weight

Learning Rate

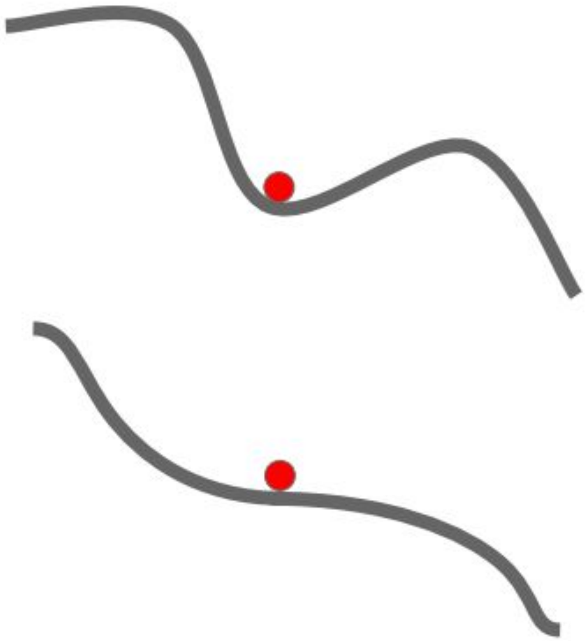
Loss function

Stochastic Gradient Descent (SGD)

Loss function is Complex



Problem with SGD



- Zero gradient
- SGD gets stuck

Momentum

Momentum

$$v_{t+1} = \rho v_t + \Delta J(w_t)$$

Gradient with Momentum

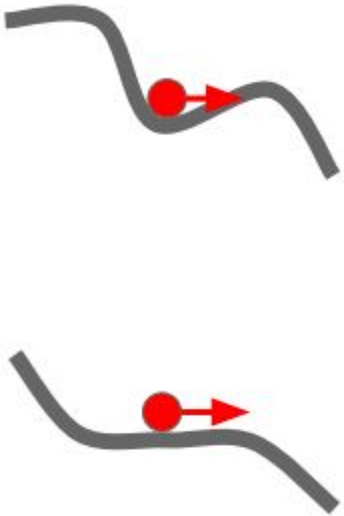
momentum

$$w_{t+1} = w_t - \alpha v_{t+1}$$

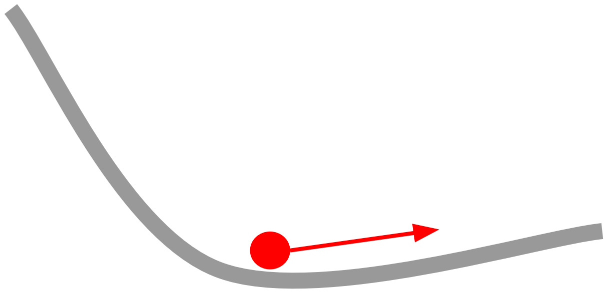
New weight

```
sgd = tf.keras.optimizers.SGD(lr=0.03, momentum=0.9)
```

What happens to Saddle point and local minima?



Can that be a problem?



Nesterov Momentum

1. Check Gradient in future
2. Adjust weights based on momentum and gradient in future

$$v_{t+1} = \rho v_t - \alpha \Delta J(w_t + \rho v_t)$$

Adjusted Momentum

$$w_{t+1} = w_t + v_{t+1}$$

```
sgd = tf.keras.optimizers.SGD(lr=0.03, momentum=0.9, nestrove=True)
```

Why should all Weights use same Learning Rate?

Adagrad

Adapts or changes learning rate for each weight

$$g_{t+1} = g_t + \Delta J(w_t)^2$$

$$w_{t+1} = w_t - \frac{\alpha \Delta J(w_t)^2}{\sqrt{g_{t+1} + \epsilon}}$$

Adagrad

1. No need to adjust learning rate

2. Learning is always decaying

```
model.compile(optimizer='adagrad', loss='categorical_crossentropy', metrics=['accuracy'])
```

Anything else we can do?

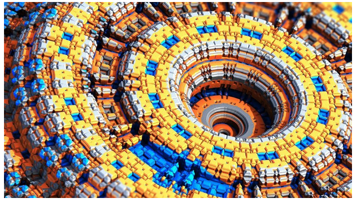
Adam

Adapts learning rate for each weight using Momentum for each weight

1. No vanishing learning rate
2. Fast convergence
3. No need to optimize learning rate manually

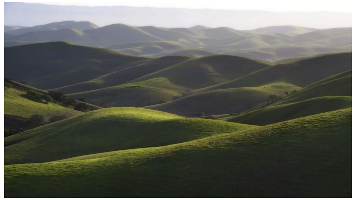
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Hyper-Parameters
in
Deep Learning



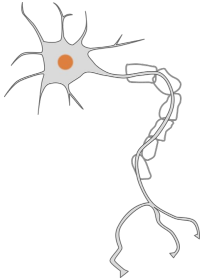
# of iterations

Batch Size

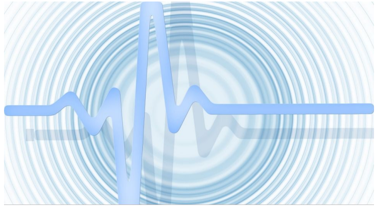


Learning Rate

# of Hidden Layers



# of Neurons in each Layer



Activation functions

Learning rate decay

Dropout

Optimizers

Maxout

$$max(w1^T + b1,\ w2^T + b2)$$

1. Generalizes ReLU and Leaky ReLU

2. Does not kill gradients

3. Compute expensive