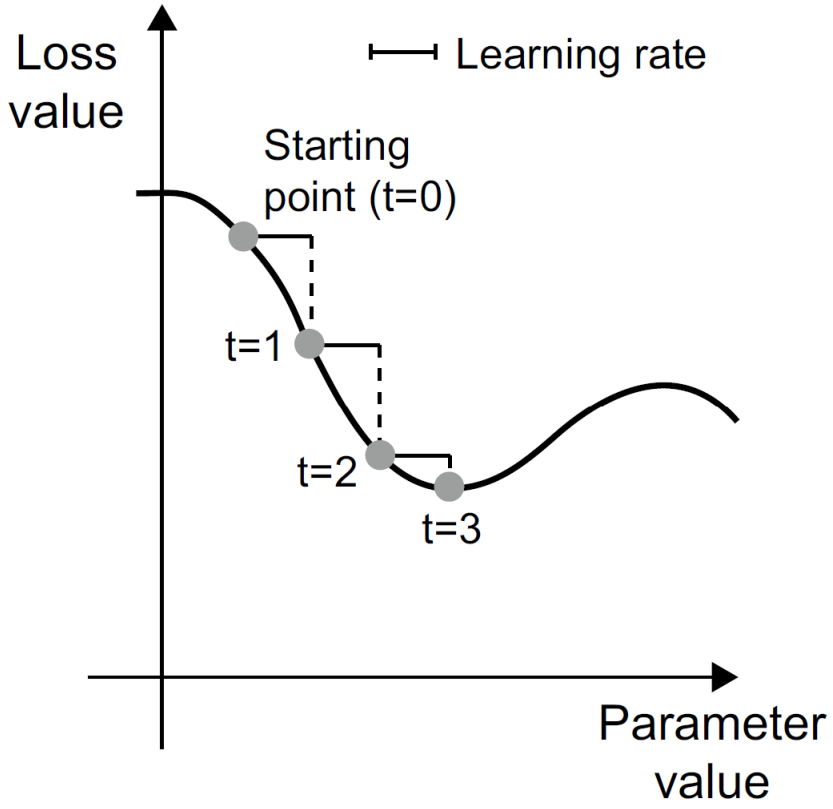
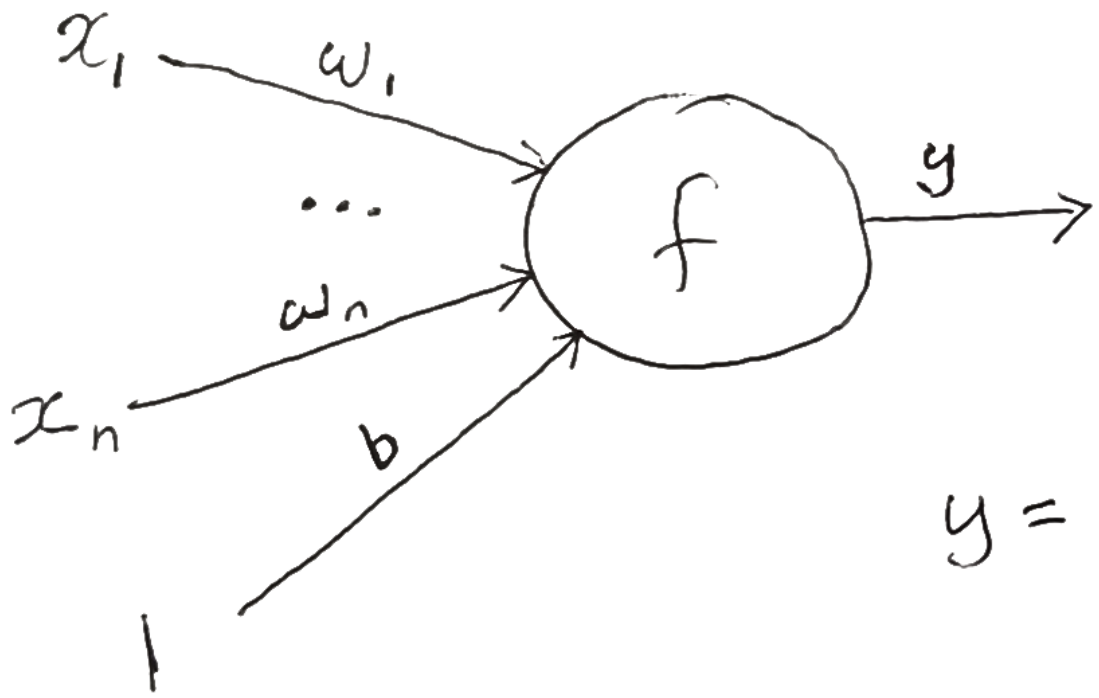
Updating Weights and Optimizers



Review: To calculate output *y* of a neuron with activation function *f* and threshold bias *b*



$$y = f\left(\sum_{i=1}^{n} x_i \cdot \omega_i + b\right)$$

How to update weights?
A simple Perceptron learning rule

input

$$\mathbf{w}'_k = \begin{cases} \mathbf{w}_k & \text{if } \mathbf{x}_k \text{ is correctly classified by } \mathbf{w}_k \\ \mathbf{w}_k - \mathbf{x}_k & \text{if false positive} \\ \mathbf{w}_k + \mathbf{x}_k & \text{if false negative} \end{cases}$$

Corrective Supervised learning!

Perceptron training algorithm (general, real valued)

- Generate random weights, usually between –0.5 & 0.5
- Repeat the following until $\boldsymbol{w}$ is found that *correctly* classifies ***all*** the training data:

  Present an item (**X**) of training data samples. The weights are modified according to the following:

$$w_i = w_i - (\alpha \cdot e \cdot x_i)$$

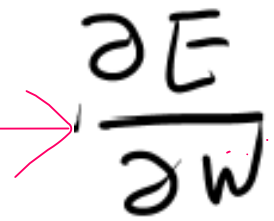
  α is the learning rate, between 0 and 1

  e = (**perceptron output - target value**)

  - 0, if output is correct: no weight change
  - Positive, if output is too high: decrease w
  - Negative, if output is too low: increase w

Mini-Batch Algorithm, Chollet's book page 52

1. Draw a batch of training samples, `x`, and corresponding targets, `y_true`.
2. Run the model on `x` to obtain predictions, `y_pred` (this is called the *forward pass*).
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Compute the gradient of the loss with regard to the model's parameters (this is called the *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example, `W -= learning_rate * gradient`—thus reducing the loss on the batch a bit. The *learning rate* (`learning_rate` here) would be a scalar factor modulating the "speed" of the gradient descent process.



Epoch, Batch Size, Iterations

- *ONE Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.*
- *Batch Size: Total number of training examples present in a single batch – "Number of samples per gradient update"*
- *Iterations is the number of batches needed to complete one epoch.*
- Example: Let's say we have 2,000 training examples. Batch size is 500. It will take 4 iterations to complete 1 epoch.

Stochastic, Mini-Batch, Batch GD

- http://adventuresinmachinelearning.com/stochastic-gradient-descent/
- Review OR, AND, and XOR problems

SGD (Stochastic Gradient Descent), batch size of 1

- Also called as iterative GD or Online GD

Error is determined for each sample

For each epoch

np.random.shuffle(X)

For each sample Xi

$$E(W) = (y_i - f_W(x_i))^2$$

When loss function is Squared Error

$$W \leftarrow W - 2\frac{\partial E}{\partial W}$$

https://sebastianraschka.com/faq/docs/mse-derivative.html

(Full) Batch Gradient Decent

Error is determined by going through all the samples

For each Epoch

$$E(w) = \sum_{i=1}^{N} (y_i - f_w(x_i))^2 \Big/ N$$

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

When loss function is MSE

Generalized abstract GD algorithm, when the full-batch size is *n*

$\hat{Y} = f_w(X)$ Model f_w's prediction with input X

$Y =$ Desired target output values

$$E = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

$$W' = W - \alpha\frac{\partial E}{\partial W}$$

Mini Batch Gradient Decent – "**Most Popular way**"

- Mini-batch gradient descent is a trade-off between stochastic gradient descent and full batch gradient descent.
- In mini-batch gradient descent, the error function (and therefore gradient) is averaged over a small number of samples, from around 10-500 or (10-1000).
- This is opposed to the SGD batch size of *1* sample, and the Full Batch GD size of *all* the training samples.

Rule of thumb to decide BS

- Small batch sizes (~32–128) are often preferred when generalization is important.
- Larger batch sizes (hundreds to thousands) are good when training speed and hardware efficiency matter, but may require learning rate scaling (e.g., linear scaling rule) to maintain performance.

Mini-Batch GD (batch size = BS)

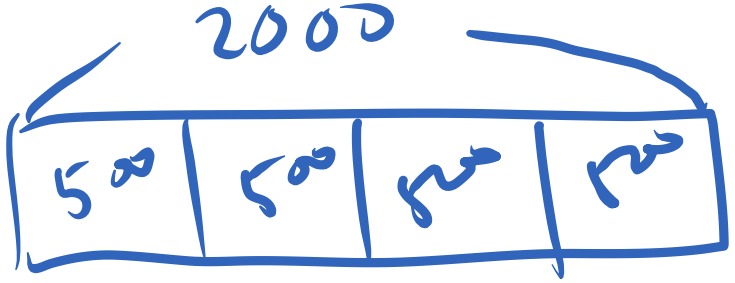
For each epoch

For each Batch

$$E(w) = \sum_{i=1}^{BS} (y_i - f_w(x_i))^2 \Big/ BS$$

$$w \leftarrow w - \alpha \cdot \frac{\partial E}{\partial w}$$

BS = 500



4 iterations

Backpropagation Algorithm

- Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the ***chain rule*** to compute the contribution that each parameter had in the loss value

- Thanks to symbolic differentiation by Tensorflow, you'll never have to implement the Backpropagation algorithm by hand.

  A symbolic differentiation program finds the derivative of a given formula with respect to a specified variable, producing a new formula as its output. In general, symbolic mathematics programs manipulate formulas to produce new formulas, rather than performing numeric calculations based on formulas. In a sense, symbolic manipulation programs are more powerful: a formula provides answers to a *class* of problems, while a numeric answer applies only to a single problem.

- All you need is an understanding of how gradient-based optimization works

Questions about MNIST code (page 62) – **Important**

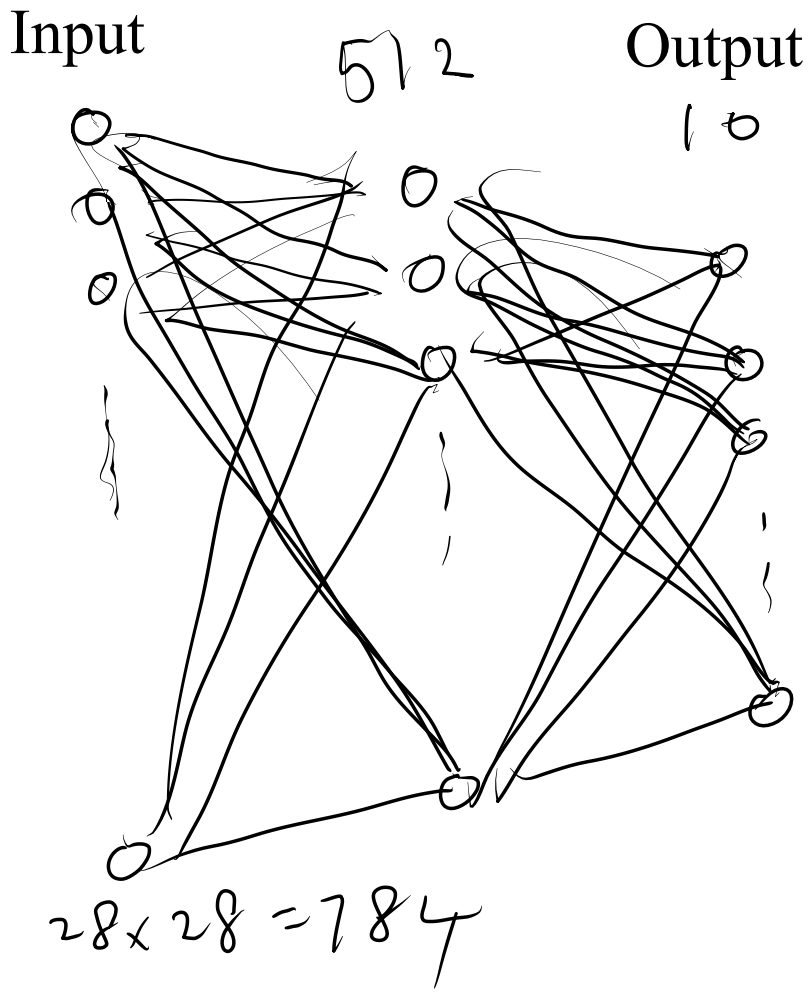
- How many trainable weights on the network? 407,050
- **How many Gradient updates were performed**?
  - There are 60,000 samples, and 128 samples in a mini batch.
  - CEIL(60,000 / 128) = 469 gradient updates (iterations) are made **per epoch**. *(Iterations is the number of batches needed to complete one epoch)*
  - 469 gradient updates/epoch * 5 epochs = 2,345 gradient updates.

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Mini-batch GD

```
Epoch 1/5
60000/60000 [==============================] - 6s 102us/step - loss: 0.2589 - acc: 0.9251
Epoch 2/5
60000/60000 [==============================] - 6s 97us/step - loss: 0.1043 - acc: 0.9686
Epoch 3/5
60000/60000 [==============================] - 7s 109us/step - loss: 0.0683 - acc: 0.9796
Epoch 4/5
60000/60000 [==============================] - 7s 120us/step - loss: 0.0505 - acc: 0.9845
Epoch 5/5
60000/60000 [==============================] - 7s 123us/step - loss: 0.0387 - acc: 0.9879
```

Review: Network model for the 1st MNIST problem: **total number of w?**



# of w

784 x 512 +
512 x 10 +
512 + 10

Vanishing gradient problem

- encountered when training artificial neural networks with gradient-based learning methods and backpropagation.
- In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. $w \leftarrow w - \alpha \frac{\partial E}{\partial w}$
- The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value.
- In the worst case, this may completely stop the neural network from further training.
- The simplest solution is to use other activation functions, such as ReLU, which doesn't cause a small derivative.

https://en.wikipedia.org/wiki/Vanishing_gradient_problem

What is Exploding Gradient?

- a challenge encountered during training deep neural networks. It occurs when the gradients of the network's loss function with respect to the weights (parameters) become excessively large.
- **Solution for Exploding Gradient Problem**
  - Weight Initialization: The weight initialization is changed to 'glorot_uniform,' which is a commonly used initialization for neural networks.
  - Gradient Clipping: The clipnorm parameter in the Adam optimizer is set to 1.0, which performs gradient clipping. This helps prevent exploding gradients.
  - Kernel Constraint: The max_norm constraint is applied to the kernel weights of each layer with a maximum norm of 2.0. This further helps in preventing exploding gradients.

https://www.geeksforgeeks.org/deep-learning/vanishing-and-exploding-gradients-problems-in-deep-learning/

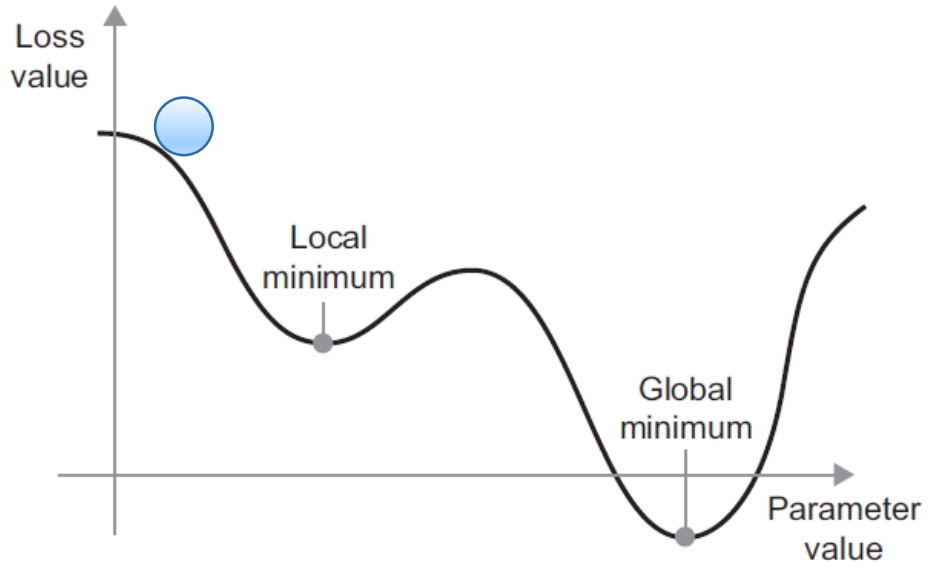
Optimization Methods in Keras, m.compile()

- https://keras.io/optimizers/
- Optimizers
  - SGD
  - rmsprop
  - adam = **rmsprop** + adaptive momentum

- How to choose Optimizer In Keras - https://www.dlology.com/blog/quick-notes-on-how-to-choose-optimizer-in-keras/

Improving GD algorithm by introducing **velocity** and **momentum**

- To avoid local minima, think of the optimization process as a small ball rolling down the loss curve. If it has enough momentum, the ball won't get stuck in a ravine and will end up at the global minimum.
- Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration).
- In practice, this means updating the parameter W based not only on the current gradient value but also on the previous parameter update, such as in a pseudo next slide.



Improving GD algorithm by introducing velocity and momentum ➔ rmsprop (root mean square propagation)

```
past_velocity = 0
momentum = 0.1
while loss is large:
    velocity = past_velocity*momentum + α*gradient
    W = W + momentum*velocity - α*gradient
    past_velocity = velocity
```

rmsprop (Cholet p. 55)

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

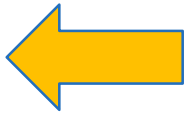
Constant momentum factor

Optimization loop

Improving GD algorithm by introducing velocity and momentum

➔ rmsprop ➔ **adam (Adaptive Moment Estimation)**

```
past_velocity = 0
momentum = 0.1
while loss is large:
    velocity = past_velocity*momentum + α*gradient
    W = W + momentum*velocity - α*gradient
    past_velocity = velocity
```



adaptive