

4. [INFO]: neural networks 101

In a nutshell

If all the terms in **bold** in the next paragraph are already known to you, you can move to the next exercise. If you are just starting in deep learning then welcome, and please read on.

A neural network classifier is made of several **layers** of **neurons**. For image classification these can be **dense** or, more frequently, **convolutional** layers. They are typically activated with the **relu** activation function. The last layer uses as many neurons as there are classes and is activated with **softmax**. For classification, **cross-entropy** is the most commonly used **loss** function, comparing the **one-hot** encoded **labels** (i.e. correct answers) with probabilities predicted by the neural network. To minimize the loss, it is best to choose an optimizer with momentum, for example **Adam** and train on **batches** of training images and labels.





For models built as a sequence of layers Keras offers the Sequential API. For example, an image classifier using three dense layers can be written in Keras as:

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28, 1]),
    tf.keras.layers.Dense(200, activation="relu"),
    tf.keras.layers.Dense(60, activation="relu"),
    tf.keras.layers.Dense(10, activation='softmax') # classifying into 10 classes
])
```

this configures the training of the model. Keras calls it "compiling" the model.

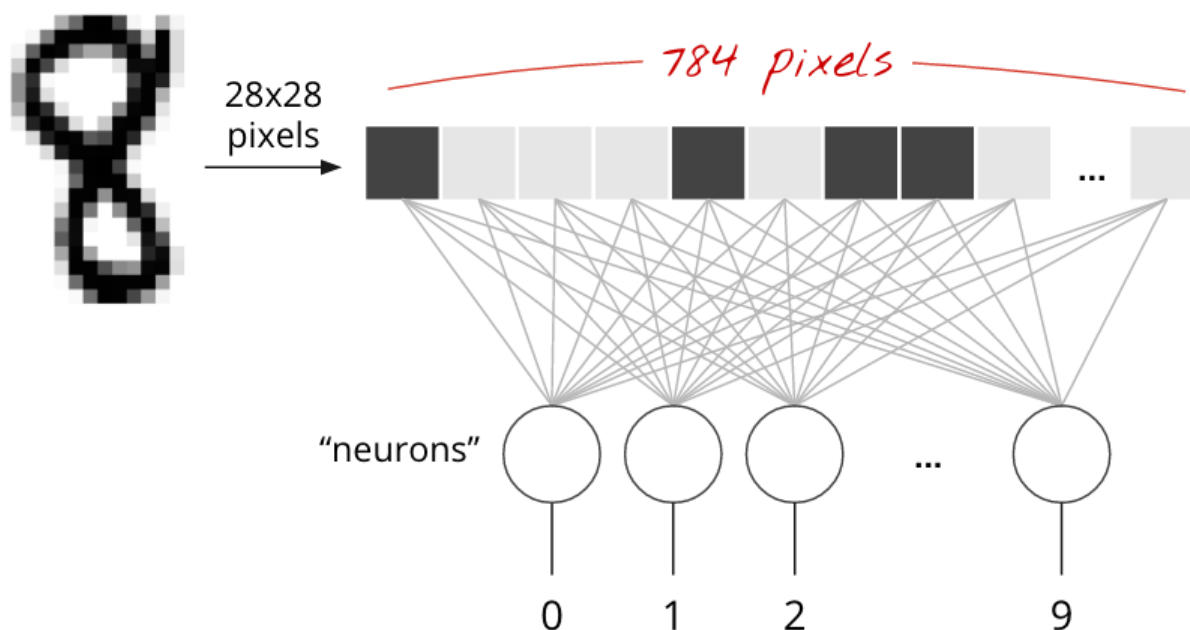
```
model.compile(
    optimizer='adam',
    loss= 'categorical_crossentropy',
    metrics=['accuracy']) # % of correct answers
```

```
# train the model
model.fit(dataset, ... )
```

If this is old news to you **SKIP**  to the next exercise otherwise **READ ON** 

A single dense layer

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images. The simplest approach for classifying them is to use the $28 \times 28 = 784$ pixels as inputs for a 1-layer neural network.



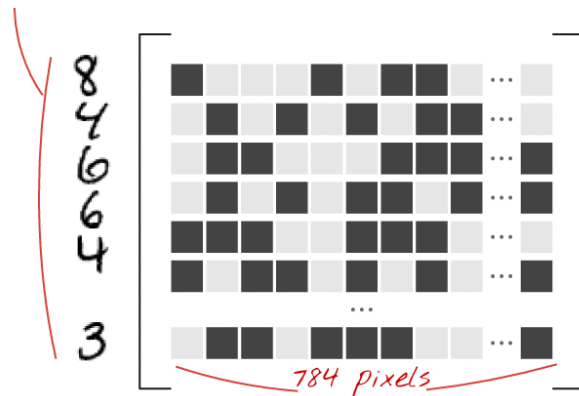
Each **"neuron"** in a neural network does a weighted sum of all of its inputs, adds a constant called the "bias" and then feeds the result through some non-linear **"activation function"**. The **"weights"** and **"biases"** are parameters that will be determined through training. They are initialized with random values at first.

The picture above represents a 1-layer neural network with 10 output neurons since we want to classify digits into 10 classes (0 to 9).

With a matrix multiplication

Here is how a neural network layer, processing a collection of images, can be represented by a matrix multiplication:

*X: 100 images,
one per line,
flattened*



10 columns

$W_{0,0}$	$W_{0,1}$	$W_{0,2}$	$W_{0,3}$...	$W_{0,9}$
$W_{1,0}$	$W_{1,1}$	$W_{1,2}$	$W_{1,3}$...	$W_{1,9}$
$W_{2,0}$	$W_{2,1}$	$W_{2,2}$	$W_{2,3}$...	$W_{2,9}$
$W_{3,0}$	$W_{3,1}$	$W_{3,2}$	$W_{3,3}$...	$W_{3,9}$
$W_{4,0}$	$W_{4,1}$	$W_{4,2}$	$W_{4,3}$...	$W_{4,9}$
$W_{5,0}$	$W_{5,1}$	$W_{5,2}$	$W_{5,3}$...	$W_{5,9}$
$W_{6,0}$	$W_{6,1}$	$W_{6,2}$	$W_{6,3}$...	$W_{6,9}$
$W_{7,0}$	$W_{7,1}$	$W_{7,2}$	$W_{7,3}$...	$W_{7,9}$
$W_{8,0}$	$W_{8,1}$	$W_{8,2}$	$W_{8,3}$...	$W_{8,9}$
...					
$W_{783,0}$	$W_{783,1}$	$W_{783,2}$...		$W_{783,9}$

784 lines

Using the first column of weights in the weights matrix W , we compute the weighted sum of all the pixels of the first image. This sum corresponds to the first neuron. Using the second column of weights, we do the same for the second neuron and so on until the 10th neuron. We can then repeat the operation for the remaining 99 images. If we call X the matrix containing our 100 images, all the weighted sums for our 10 neurons, computed on 100 images are simply $X.W$, a matrix multiplication.

Each neuron must now add its bias (a constant). Since we have 10 neurons, we have 10 bias constants. We will call this vector of 10 values b . It must be added to each line of the previously computed matrix. Using a bit of magic called "broadcasting" we will write this with a simple plus sign.

"Broadcasting" is a standard trick used in Python and numpy, its scientific computation library. It extends how normal operations work on matrices with incompatible dimensions. "Broadcasting add" means "if you are adding two matrices but you cannot because their dimensions are not compatible, try to replicate the small one as much as needed to make it work."

We finally apply an activation function, for example "softmax" (explained below) and obtain the formula describing a 1-layer neural network, applied to 100 images:

Predictions
 $Y[100, 10]$

Images
 $X[100, 784]$

Weights
 $W[784, 10]$

Biases
 $b[10]$

$$Y = \text{softmax}(X.W + b)$$

applied line by line
matrix multiply
broadcast on all lines

tensor shapes in []

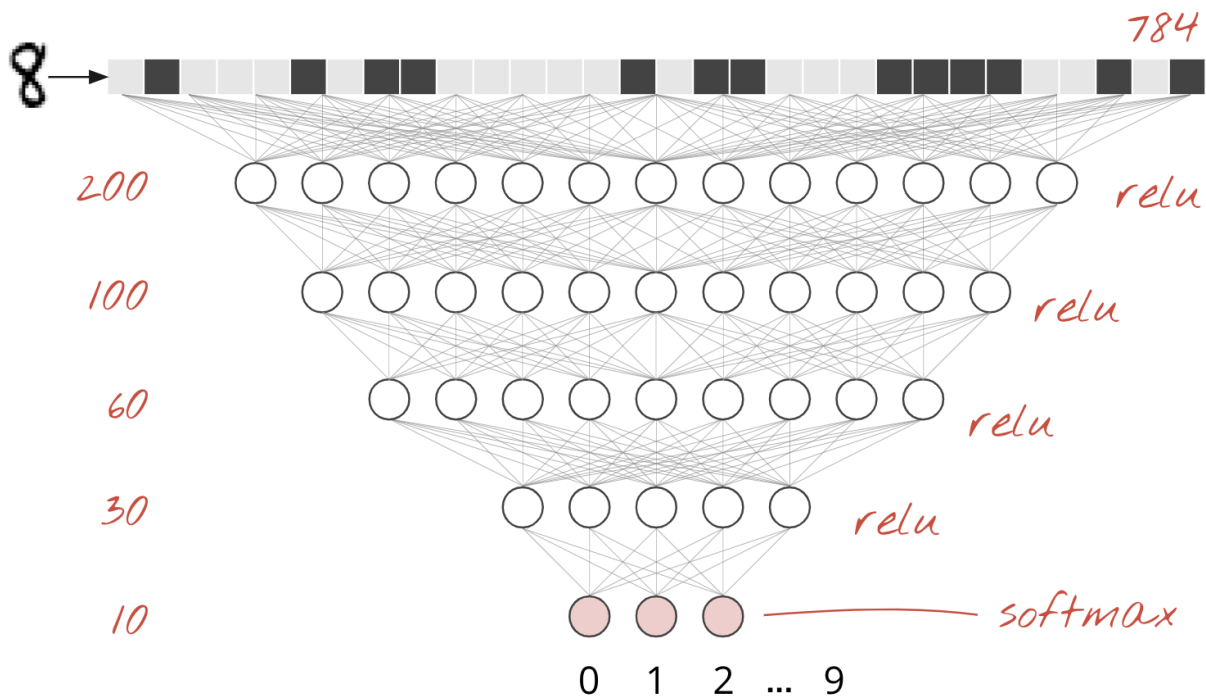
In Keras

With high-level neural network libraries like Keras, we will not need to implement this formula. However, it is important to understand that a neural network layer is just a bunch of multiplications and additions. In Keras, a dense layer would be written as:

```
tf.keras.layers.Dense(10, activation='softmax')
```

Go deep

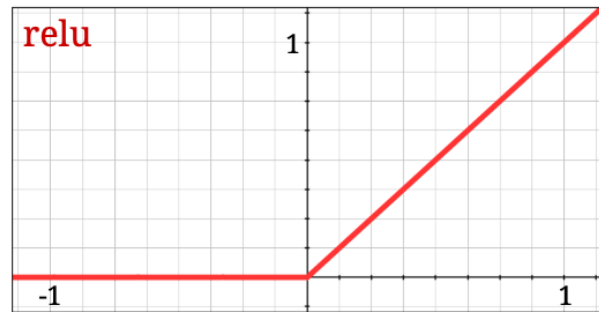
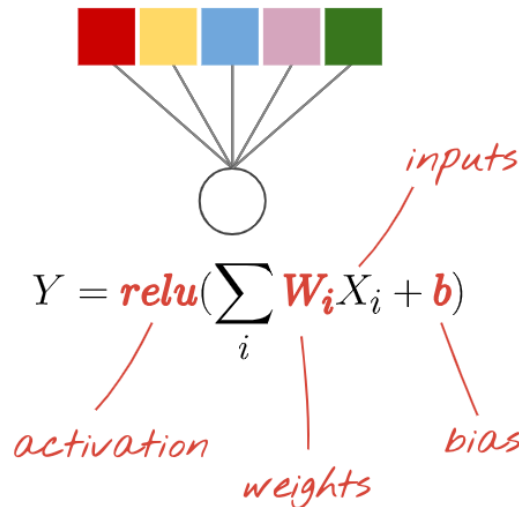
It is trivial to chain neural network layers. The first layer computes weighted sums of pixels. Subsequent layers compute weighted sums of the outputs of the previous layers.



The only difference, apart from the number of neurons, will be the choice of activation function.

Activation functions: relu, softmax and sigmoid

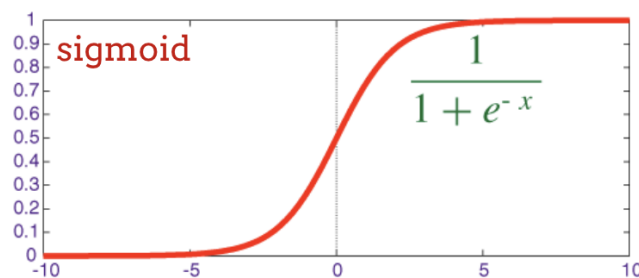
You would typically use the "relu" activation function for all layers but the last. The last layer, in a classifier, would use "softmax" activation.



Again, a "neuron" computes a weighted sum of all of its inputs, adds a value called "bias" and feeds the result through the activation function.

The most popular activation function is called "**RELU**" for Rectified Linear Unit. It is a very simple function as you can see on the graph above.

The traditional activation function in neural networks was the "**sigmoid**" but the "relu" was shown to have better convergence properties almost everywhere and is now preferred.



Softmax activation for classification

The last layer of our neural network has 10 neurons because we want to classify handwritten digits into 10 classes (0,...9). It should output 10 numbers between 0 and 1 representing the probability of this digit being a 0, a 1, a 2 and so on. For this, on the last layer, we will use an activation function called "**softmax**".

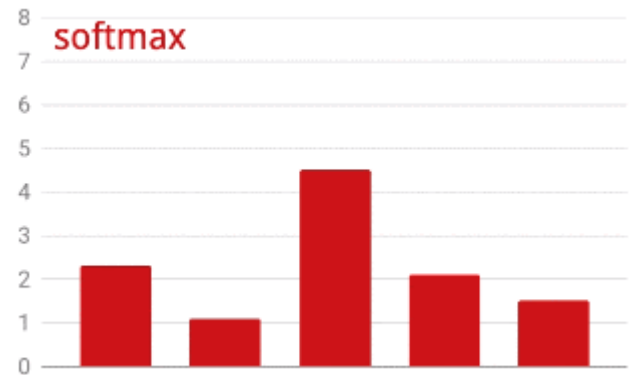
Applying softmax on a vector is done by taking the exponential of each element and then normalising the vector, typically by dividing it by its "L1" norm (i.e. sum of absolute values) so that normalized values add up to 1 and can be interpreted as probabilities.

The output of the last layer, before activation is sometimes called "**logits**". If this vector is $L = [L_0, L_1, L_2, L_3, L_4, L_5, L_6, L_7, L_8, L_9]$, then:

$$\text{softmax}(L_n) = \frac{e^{L_n}}{\|e^L\|}$$

weighted sum+bias (pointing to L_n)

L1 norm (pointing to $\|e^L\|$)



Why is "softmax" called softmax? The exponential is a steeply increasing function. It will increase differences between the neuron outputs. Then, as you normalise the vector, the largest element, which dominates the norm, will be normalised to a value close to 1 while all the other elements will end up divided by a large value and normalised to something close to 0. The resulting vector clearly shows which is the winning class, the "max", but retains the original relative order of its values, hence the "soft".

Cross-entropy loss

Now that our neural network produces predictions from input images, we need to measure how good they are, i.e. the distance between what the network tells us and the correct answers, often called "labels". Remember that we have correct labels for all the images in the dataset.

Any distance would work, but for classification problems the so-called "cross-entropy distance" is the [most effective](#). We will call this our error or "loss" function:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	0	0	0

actual probabilities, "one-hot" encoded

Cross entropy: $-\sum Y'_i \cdot \log(Y_i)$

computed probabilities

.01	.03	.00	.04	.03	.05	.8	.02	.01	.01
0	1	2	3	4	5	6	7	8	9

this is a "6"

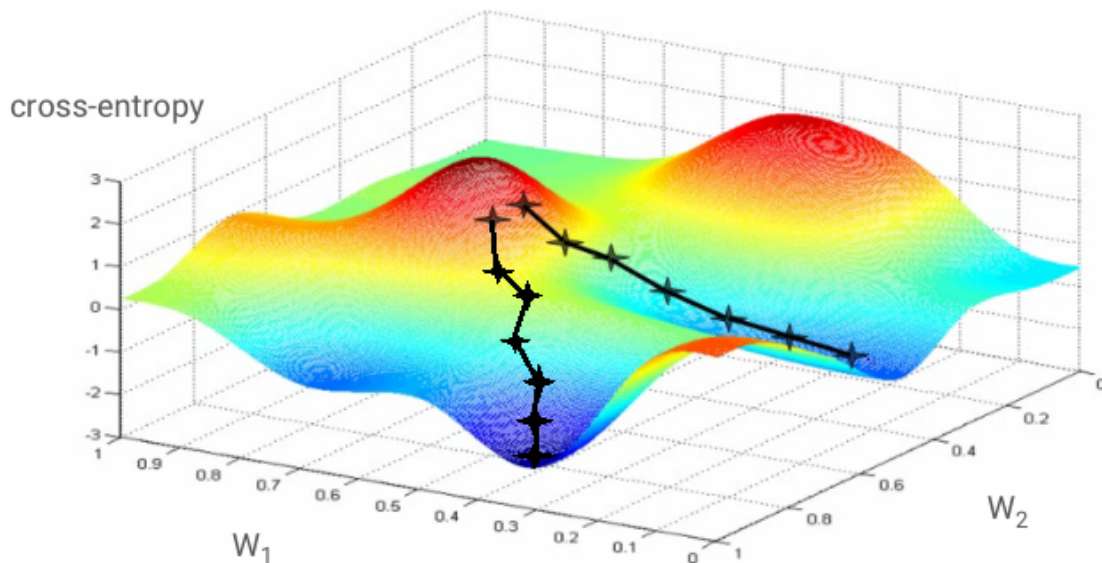
"One-hot" encoding means that you represent the label "this is the digit 3" by using a vector of 10 values, all zeros except for the 3rd value which is 1. This vector represents a 100% probability of being the "digit 3". Our neural network also outputs its predictions as a vector of 10 probability values. They are easy to compare.

Gradient descent

"Training" the neural network actually means using training images and labels to adjust weights and biases so as to minimise the cross-entropy loss function. Here is how it works.

The cross-entropy is a function of weights, biases, pixels of the training image and its known class.

If we compute the partial derivatives of the cross-entropy relatively to all the weights and all the biases we obtain a "gradient", computed for a given image, label, and present value of weights and biases. Remember that we can have millions of weights and biases so computing the gradient sounds like a lot of work. Fortunately, TensorFlow does it for us. The mathematical property of a gradient is that it points "up". Since we want to go where the cross-entropy is low, we go in the opposite direction. We update weights and biases by a fraction of the gradient. We then do the same thing again and again using the next batches of training images and labels, in a training loop. Hopefully, this converges to a place where the cross-entropy is minimal although nothing guarantees that this minimum is unique.



"Learning rate": You cannot update your weights and biases by the whole length of the gradient at each iteration. It would be like trying to get to the bottom of a valley while wearing seven-league boots. You would be jumping from one side of the valley to the other. To get to the bottom, you need to do smaller steps, i.e. use only a fraction of the gradient, typically in the 1/1000th range. This fraction is called the "learning rate".

Mini-batching and momentum

You can compute your gradient on just one example image and update the weights and biases immediately, but doing so on a batch of, for example, 128 images gives a gradient that better represents the constraints imposed by different example images and is therefore likely to converge towards the solution faster. The size of the mini-batch is an adjustable parameter.

This technique, sometimes called "stochastic gradient descent" has another, more pragmatic benefit: working with batches also means working with larger matrices and these are usually easier to optimise on GPUs and TPUs.

The convergence can still be a little chaotic though and it can even stop if the gradient vector is all zeros. Does that mean that we have found a minimum? Not always. A gradient component can be zero on a minimum or a maximum. With a gradient vector with millions of elements, if they are all zeros, the probability that every zero corresponds to a minimum and none of them to a maximum point is pretty small. In a space of many dimensions, saddle points are pretty common and we do not want to stop at them.

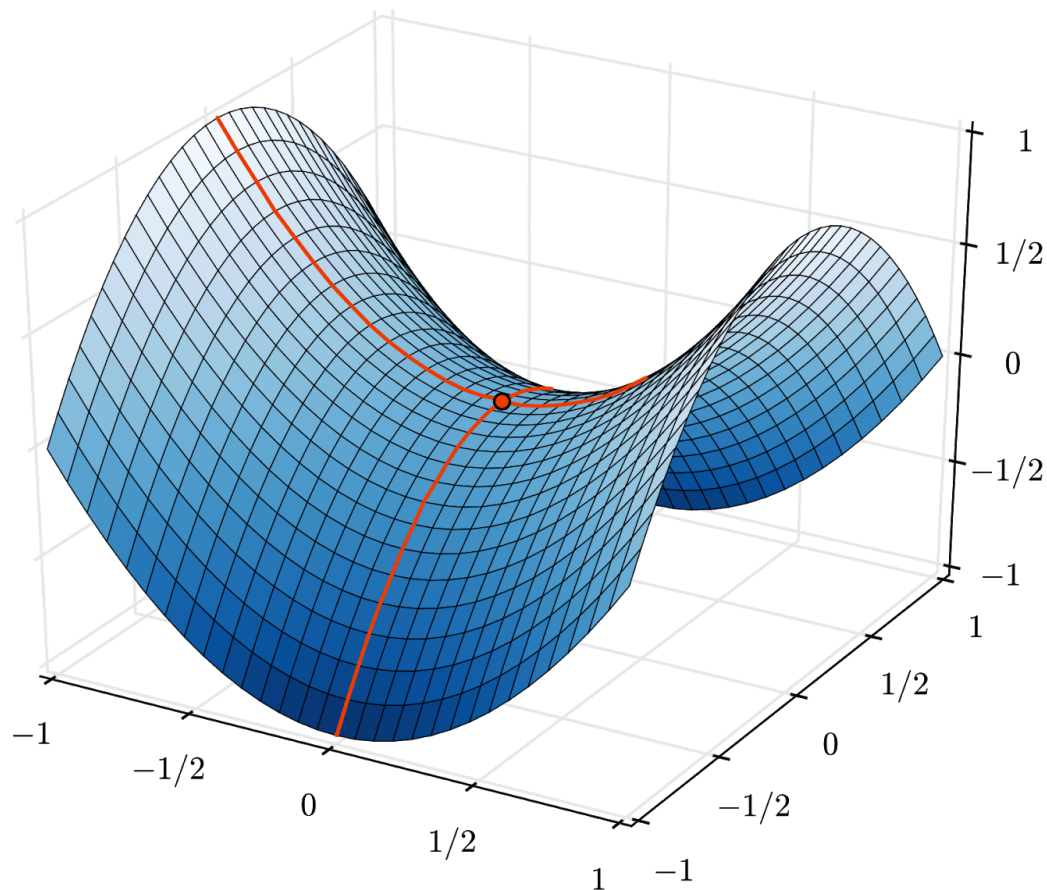


Illustration: a saddle point. The gradient is 0 but it is not a minimum in all directions. (Image attribution [Wikimedia: By Nicoguaro - Own work, CC BY 3.0](#))

The solution is to add some momentum to the optimization algorithm so that it can sail past saddle points without stopping.

The TensorFlow library provides a whole range of optimizers, starting with basic gradient descent `tf.keras.optimizers.SGD`, which now has an optional `momentum` parameter. More advanced popular optimizers that have a built-in momentum are `tf.keras.optimizers.RMSprop` or `tf.keras.optimizers.Adam`.

Glossary

batch or **mini-batch**: training is always performed on batches of training data and labels. Doing so helps the algorithm converge. The "batch" dimension is typically the first dimension of data tensors. For example a tensor of shape `[100, 192, 192, 3]` contains 100 images of 192x192 pixels with three values per pixel (RGB).

cross-entropy loss: a special loss function often used in classifiers.

dense layer: a layer of neurons where each neuron is connected to all the neurons in the previous layer.

features: the inputs of a neural network are sometimes called "features". The art of figuring out which parts of a dataset (or combinations of parts) to feed into a neural network to get good predictions is called "feature engineering".

labels: another name for "classes" or correct answers in a supervised classification problem

learning rate: fraction of the gradient by which weights and biases are updated at each iteration of the training loop.

logits: the outputs of a layer of neurons before the activation function is applied are called "logits". The term comes from the "logistic function" a.k.a. the "sigmoid function" which used to be the most popular activation function. "Neuron outputs before logistic function" was shortened to "logits".

loss: the error function comparing neural network outputs to the correct answers

neuron: computes the weighted sum of its inputs, adds a bias and feeds the result through an activation function.

one-hot encoding: class 3 out of 5 is encoded as a vector of 5 elements, all zeros except the 3rd one which is 1.

relu: rectified linear unit. A popular activation function for neurons.

sigmoid: another activation function that used to be popular and is still useful in special cases.

softmax: a special activation function that acts on a vector, increases the difference between the largest component and all others, and also normalizes the vector to have a sum of 1 so that it can be interpreted as a vector of probabilities. Used as the last step in classifiers.

tensor: A "tensor" is like a matrix but with an arbitrary number of dimensions. A 1-dimensional tensor is a vector. A 2-dimensions tensor is a matrix. And then you can have tensors with 3, 4, 5 or more dimensions.