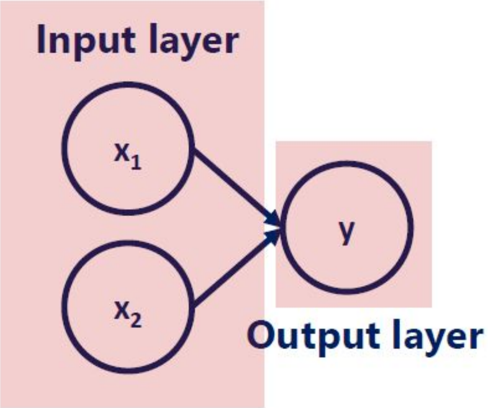
Layers

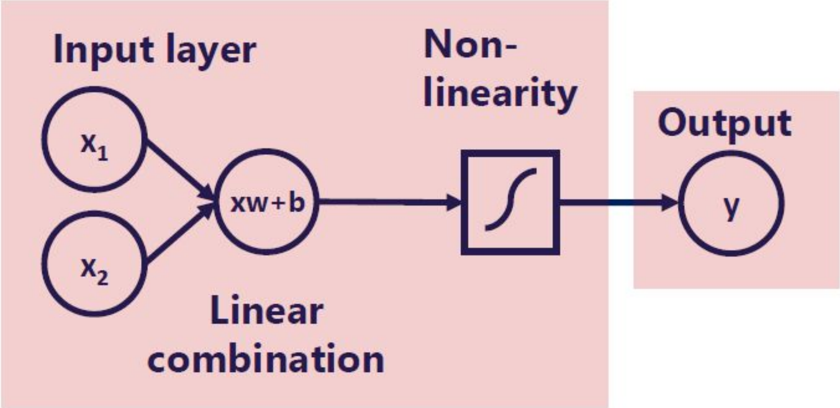
An initial linear combination and the added non-linearity form a **layer**. The layer is the building block of neural networks.

Minimal example (a simple neural network)



In the minimal example we trained a *neural network* which had no depth. There were solely an input layer and an output layer. Moreover, the output was simply **a linear combination** of the input.

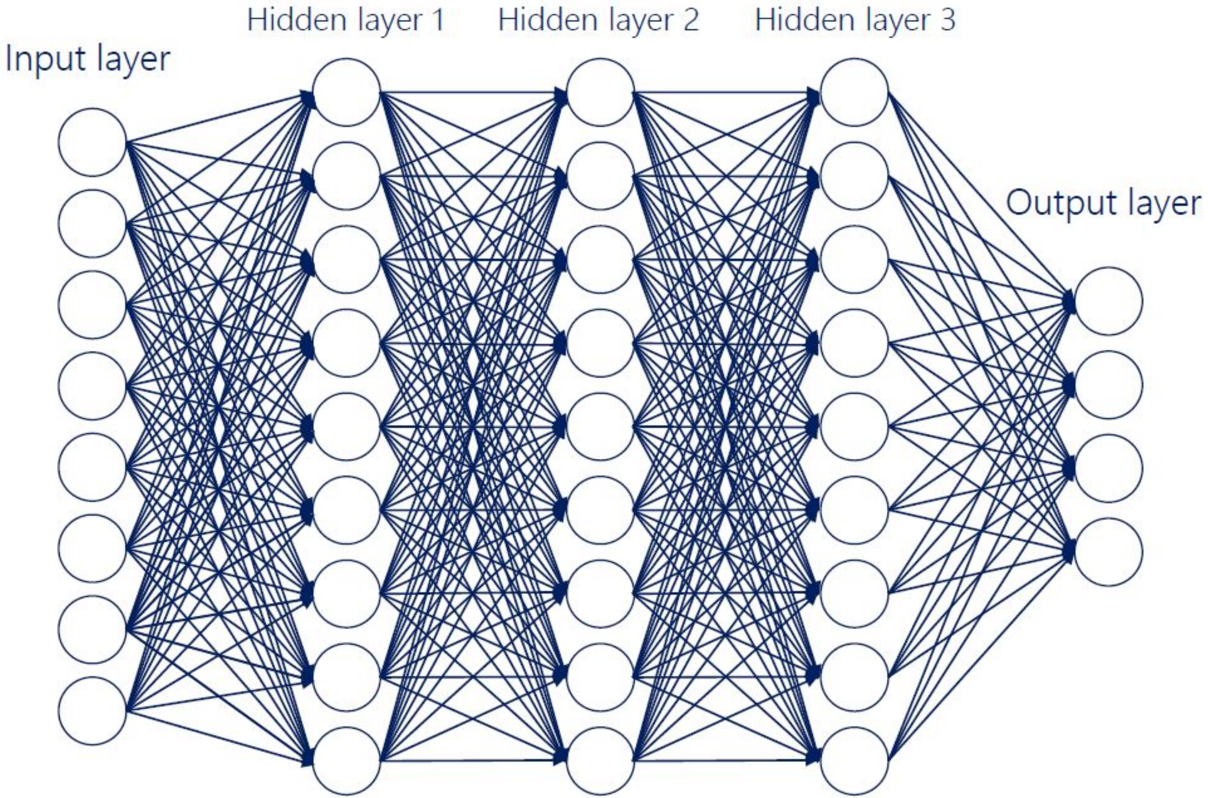
Neural networks



Neural networks step on linear combinations, but add a non-linearity to each one of them. Mixing linear combinations and non-linearities allows us to model arbitrary functions.

A deep net

This is a deep neural network (deep net) with 5 layers.



How to read this diagram:

A layer

A unit (a neuron)

Arrows represent
mathematical transformations

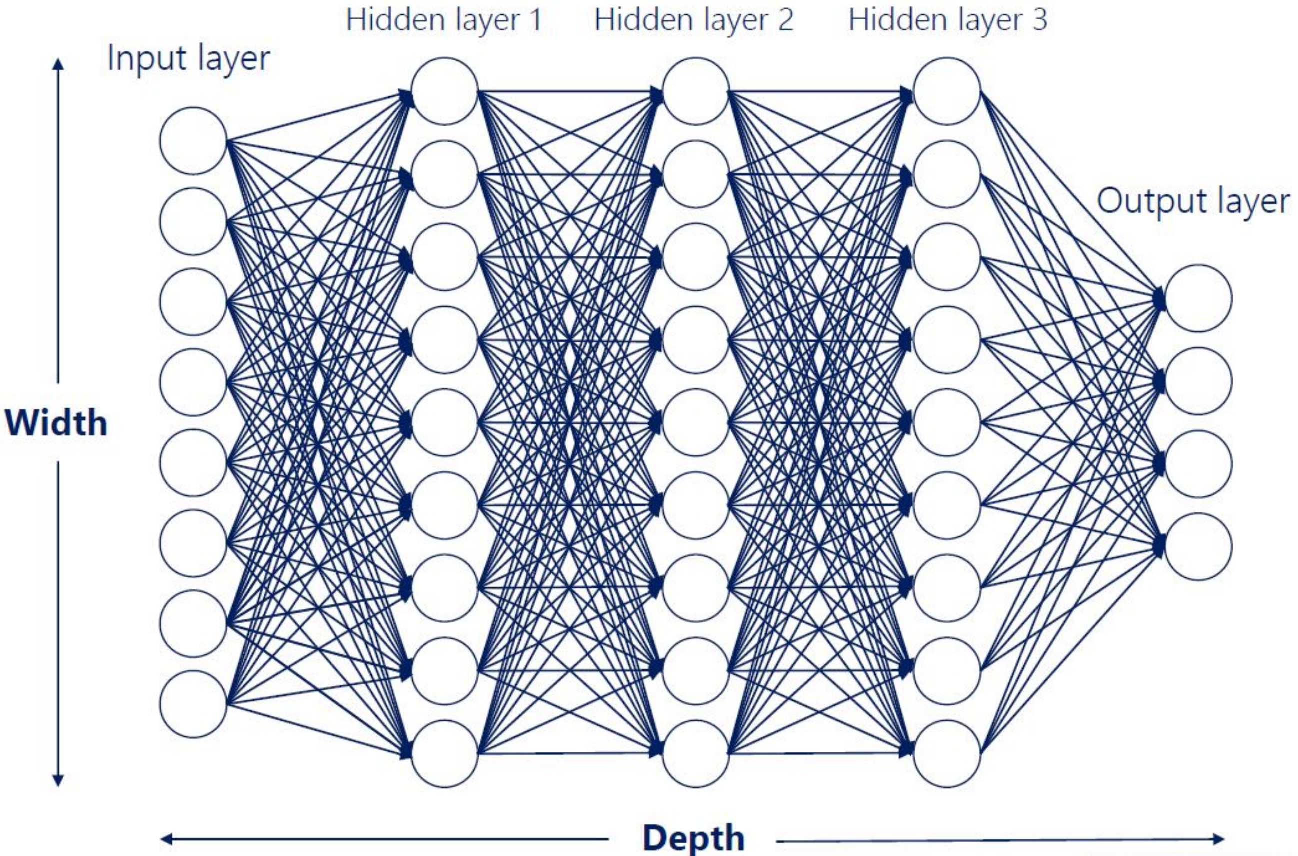
A deep net

The **width** of a layer is the number of units in that layer

The **width** of the net is the number of units of the biggest layer

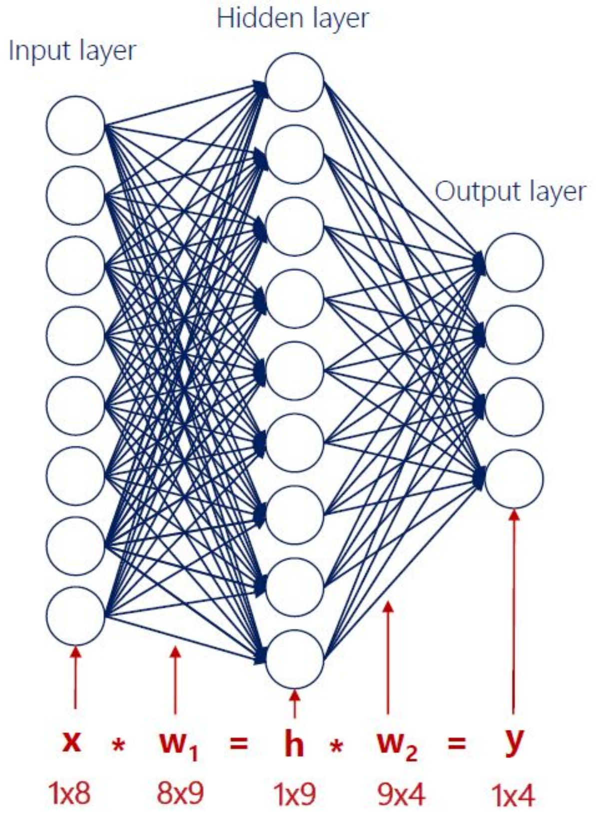
The **depth** of the net is equal to the number of layers or the number of hidden layers. The term has different definitions. More often than not, we are interested in the number of hidden layers (as there are always input and output layers).

The width and the depth of the net are called **hyperparameters.** They are values we manually chose when creating the net.



Why we need non-linearities to stack layers

You can see a net with no non-linearities: just linear combinations.



$$h = x * w_1$$

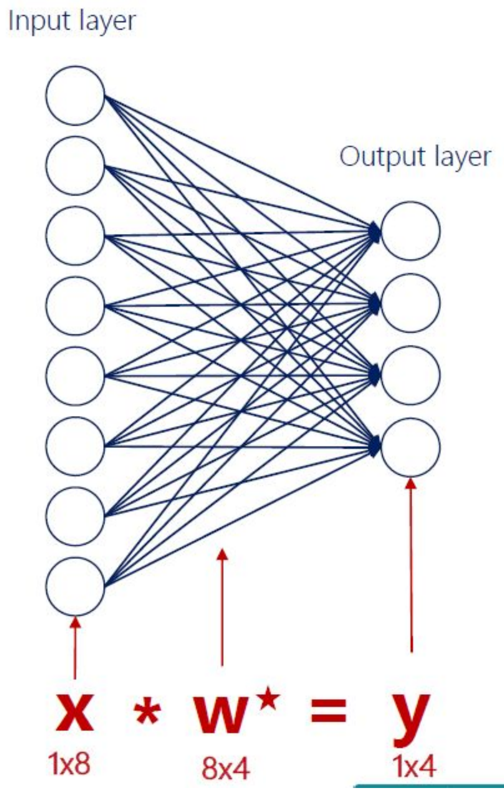
$$y = h * w_2$$

$$y = x * \boxed{\underset{8\text{x}9}{w_1} * \underset{9\text{x}4}{w_2}}$$

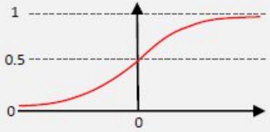
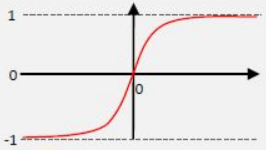
$$y = x * \underset{8\text{x}4}{w^\star}$$

Two consecutive linear transformations are equivalent to a single one.

Two consecutive linear transformations are equivalent to a single one.

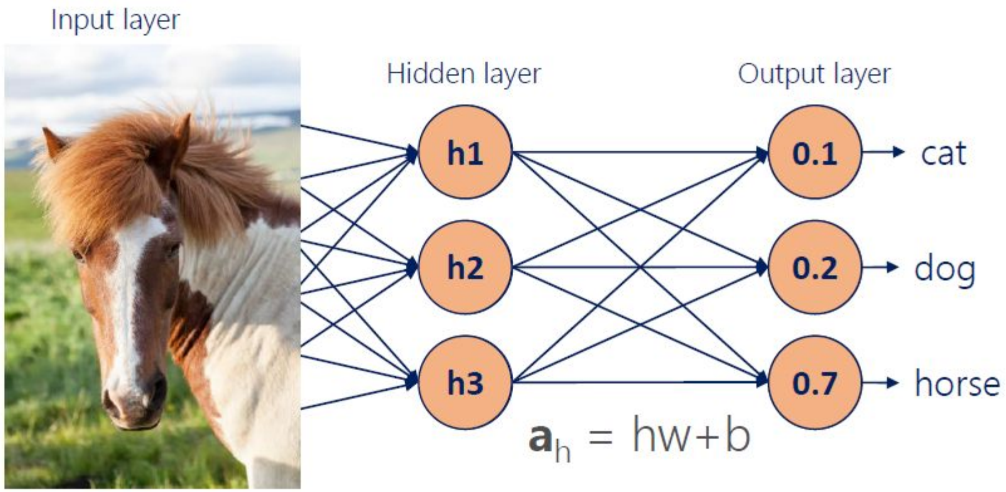


Common activation functions

Name	Formula	Derivative	Graph	Range
sigmoid (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$		(0,1)
TanH (hyperbolic tangent)	$\tanh(a) = \frac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^{a} + e^{-a})^2}$		(-1,1)
ReLu (rectified linear unit)	relu(a) = max(0,a)	$\frac{\partial \operatorname{relu}(a)}{\partial a} = \begin{cases} 0, if\ a \le 0 \\ 1, if\ a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_{\mathrm{i}}(\boldsymbol{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_{\mathrm{i}}(\boldsymbol{a})}{\partial a_j} = \sigma_i(\boldsymbol{a})\left(\delta_{ij} - \sigma_j(\boldsymbol{a})\right)$ Where δ_{ij} is 1 if i=j, 0 otherwise		(0,1)

All common activation functions are: **monotonic, continuous,** and **differentiable**. These are important properties needed for the optimization.

Softmax activation



The softmax activation transforms a bunch of arbitrarily large or small numbers into a valid probability distribution.

While other activation functions get an input value and transform it, regardless of the other elements, the softmax considers the information about the **whole set of numbers** we have.

The values that softmax outputs are in the range from 0 to 1 and their sum is exactly 1 (like probabilities).

Example:

$\mathbf{a}$ = [-0.21,0.47,1.72]

$$\text{softmax}(\mathbf{a}) = \frac{e^{a_i}}{\sum_j e^{a_j}}$$

$$\sum_j e^{a_j} = e^{-0.21} + e^{0.47} + e^{1.72} = 8$$

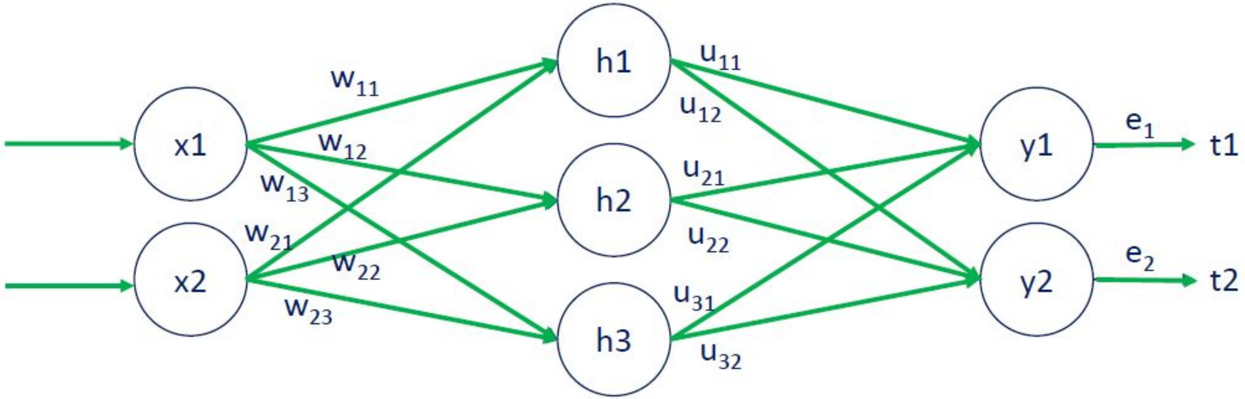
$$\text{softmax}(\mathbf{a}) = \left[\frac{e^{-0.21}}{8}, \frac{e^{0.47}}{8}, \frac{e^{1.72}}{8}\right]$$

y = [0.1,0.2,0.7] → probability distribution

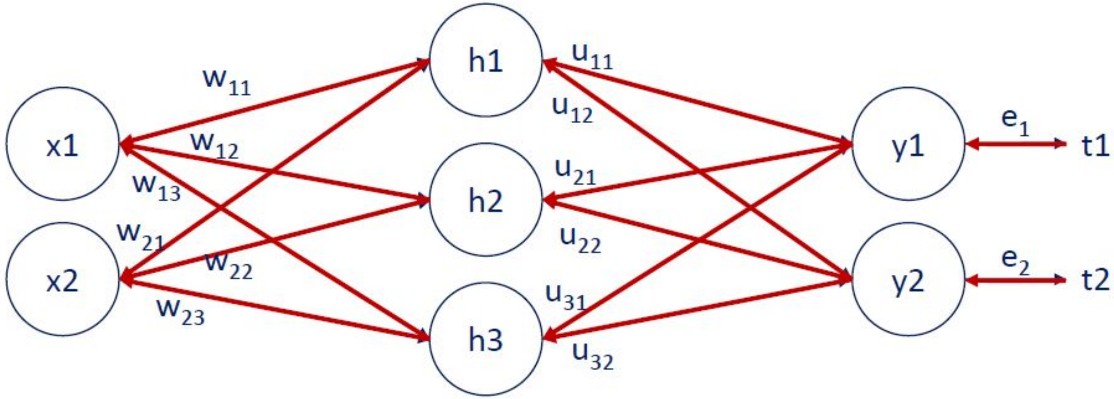
The property of the softmax to output probabilities is so useful and intuitive that it is often used as the activation function for the **final (output) layer.**

However, when the softmax is used prior to that (as the activation of a hidden layer), the results are not as satisfactory. That's because a lot of the information about the variability of the data is lost.

Backpropagation



Forward propagation is the process of pushing inputs through the net. At the end of each epoch, the obtained outputs are compared to targets to form the errors.



Backpropagation of errors is an **algorithm** for neural networks using gradient descent. It consists of calculating the contribution of each **parameter** to the errors. We backpropagate the **errors** through the net and **update** the parameters (weights and biases) accordingly.

The objective function is a measure of how well our model's outputs match the targets.

The **targets** are the "correct values" which we aim at. In the cats and dogs example, the targets were the "labels" we assigned to each photo (either "cat" or "dog").

Objective functions can be split into two types: **loss** (supervised learning) and **reward** (reinforcement learning). Our focus is supervised learning.

Common loss functions

L2-norm loss
/regression/

$$\sum_i (y_i - t_i)^2$$

The L2-norm of a vector, **a**, (Euclidean length) is given by

$$\|a\| = \sqrt{a^T \cdot a} = \sqrt{a_1^2 + \cdots + a_n^2}$$

The main rationale is that the L2-norm loss is basically the distance from the origin (0). So, the closer to the origin is the difference of the outputs and the targets, the lower the loss, and the better the prediction.

Cross-entropy loss
/classification/

$$-\sum_i \boldsymbol{t}_i \ln \boldsymbol{y}_i$$

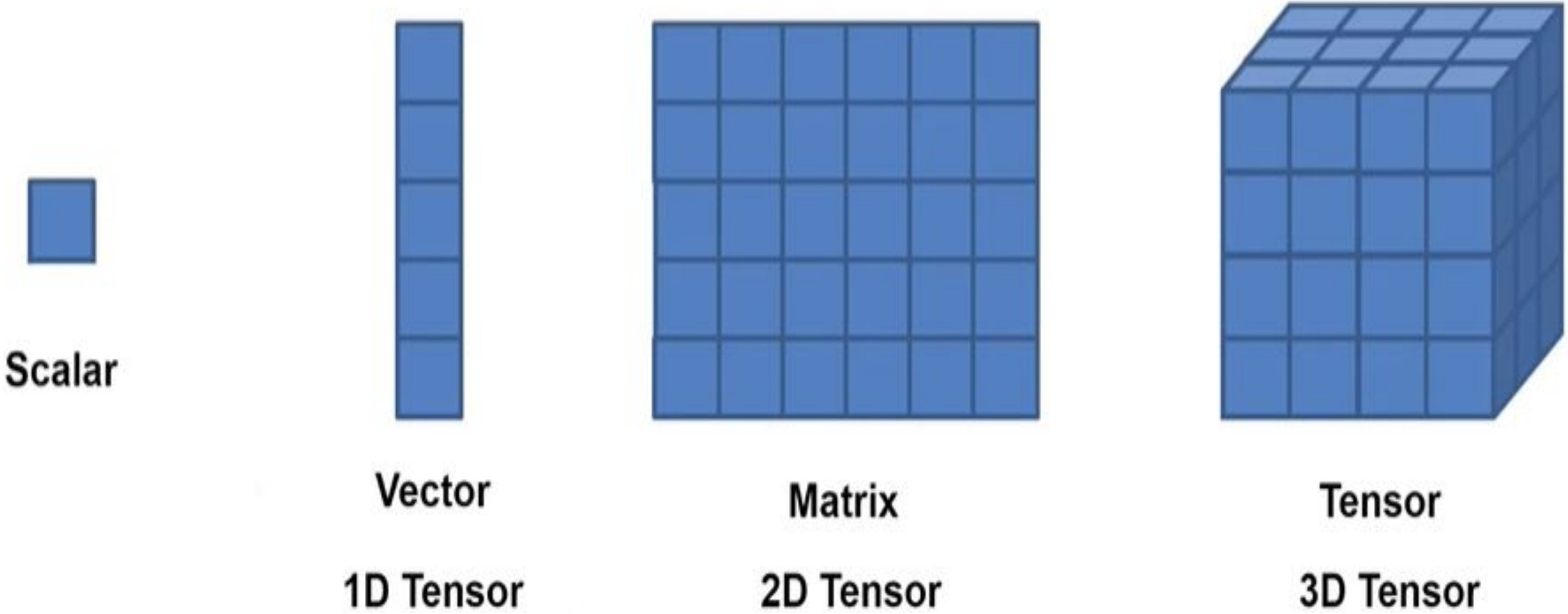
The cross-entropy loss is mainly used for classification. Entropy comes from information theory, and measures how much information one is missing to answer a question. Cross-entropy (used in ML) works with probabilities – one is our opinion, the other – the true probability (the probability of a target to be correct is 1 by definition). If the cross-entropy is 0, then we are **not missing any information** and have a perfect model.

- Let’s begin by learning how to use PyTorch as a tensor array library.
- What is a tensor?
    - A tensor is often thought of as a generalized matrix.
    - You can have 1-D, 2-D, 3-D, N-D tensors!

Scalar | Vector | Matrix | Tensor

$$1 \qquad \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad \begin{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} & \begin{bmatrix} 3 & 2 \end{bmatrix} \\ \begin{bmatrix} 1 & 7 \end{bmatrix} & \begin{bmatrix} 5 & 4 \end{bmatrix} \end{bmatrix}$$

- Often these lower dimensional tensors have specific names:



```
tensor_2D = torch.tensor([ [[1,2,3],[4,5,6],[7,8,9],[10,11,12]] ])
print(tensor_2D)
print(tensor_2D.shape)
```

```
tensor([[[ 1,  2,  3],
         [ 4,  5,  6],
         [ 7,  8,  9],
         [10, 11, 12]]])

torch.Size([1, 4, 3])
```

```
tensor_3D = torch.tensor([ [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                           [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                           [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ]
                         ])
print(tensor_3D)
tensor_3D.shape
```

```
tensor([[[ 1,  2,  3],
         [ 4,  5,  6],
         [ 7,  8,  9],
         [10, 11, 12]],

        [[ 1,  2,  3],
         [ 4,  5,  6],
         [ 7,  8,  9],
         [10, 11, 12]],

        [[ 1,  2,  3],
         [ 4,  5,  6],
         [ 7,  8,  9],
         [10, 11, 12]]])

torch.Size([3, 4, 3])
```

```python
tensor_3D_extended = torch.tensor([ [ [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] ],
                                      [
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] ],
                                      [
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ],
                                        [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] ]
                                  ])
tensor_3D_extended.shape
```

```
torch.Size([3, 3, 4, 3])
```

```python
import torch
import matplotlib.pyplot as plt
import numpy as np
import torch.nn.functional as F
from torch import nn
from torchvision import datasets, transforms
```

```python
transform = transforms.Compose([transforms.Resize((28,28)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))
                                ])


training_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
validation_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

training_loader = torch.utils.data.DataLoader(training_dataset, batch_size=100, shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size = 100, shuffle=False)
```

```python
for x, target in iter(training_loader):
    print(target)
    print(x.shape)
    break
```

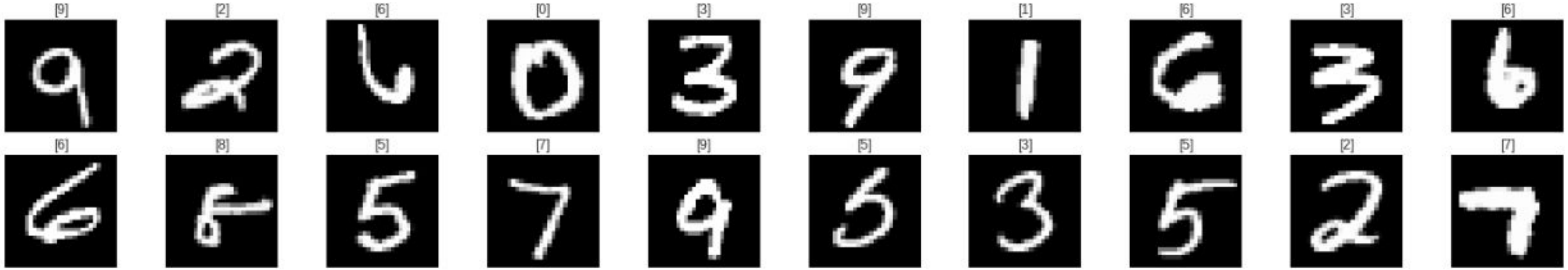
```
tensor([1, 2, 7, 1, 8, 2, 1, 1, 5, 6, 6, 0, 2, 2, 5, 7, 4, 1, 8, 3, 2, 8, 6, 8,
        1, 1, 9, 7, 1, 9, 0, 0, 5, 0, 1, 2, 3, 6, 9, 5, 2, 1, 4, 8, 9, 9, 2, 7,
        6, 5, 2, 4, 7, 2, 8, 6, 7, 5, 1, 1, 1, 8, 1, 5, 9, 6, 3, 5, 4, 0, 8, 6,
        6, 2, 6, 2, 1, 4, 6, 7, 0, 8, 0, 5, 4, 8, 9, 9, 5, 6, 4, 1, 2, 1, 7, 3,
        7, 6, 8, 4])
torch.Size([100, 1, 28, 28])
```

```python
for x, target in iter(validataion_loader):
    print(target.shape)
    print(x.shape)
    break
```

```
torch.Size([100])
torch.Size([100, 1, 28, 28])
```

```python
def im_convert(tensor):
    image = tensor.clone().detach().numpy()
    image = image.transpose(1, 2, 0)     # 1,2, vor lini 28x28x1 shape i nkar isk mer mot 1x28x28 er indexnerneq poxum
    image = image * np.array((0.5, 0.5, 0.5)) + np.array((0.5, 0.5, 0.5))  # denormalization part na es mean = 0.5 std = 0.5
    image = image.clip(0, 1)             # clip from [-1,1] to [0,1]
    return image
```

```python
dataiter = iter(training_loader)
images, labels = dataiter.next()
fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
    ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
    plt.imshow(im_convert(images[idx]))
    ax.set_title([labels[idx].item()])
```



```python
class Classifier(nn.Module):

    def __init__(self, D_in, H1, H2, D_out):
        super().__init__()
        self.linear1 = nn.Linear(D_in, H1)
        self.linear2 = nn.Linear(H1, H2)
        self.linear3 = nn.Linear(H2, D_out)
    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = F.softmax(self.linear3(x))
        return x
```

```python
model = Classifier(784, 125, 65, 10)
model
```

```
Classifier(
  (linear1): Linear(in_features=784, out_features=125, bias=True)
  (linear2): Linear(in_features=125, out_features=65, bias=True)
  (linear3): Linear(in_features=65, out_features=10, bias=True)
)
```

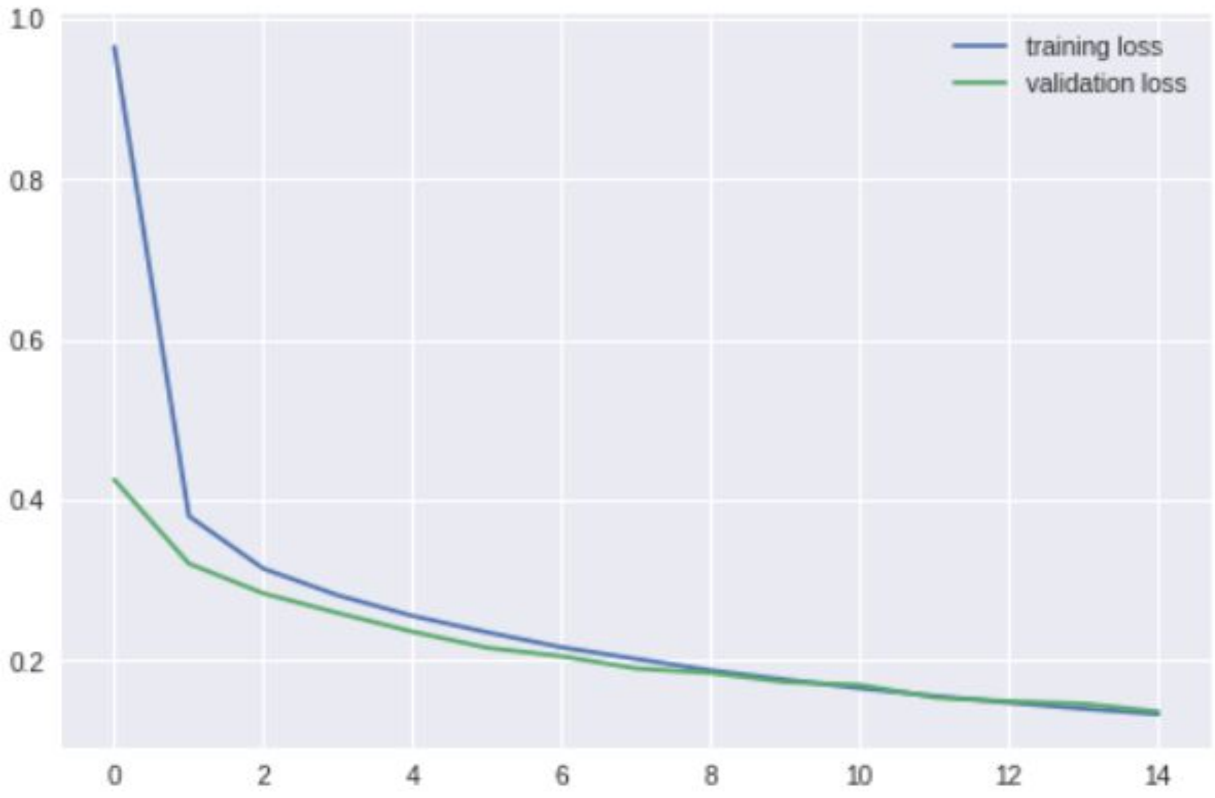
```python
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.0001)
```

```python
epochs = 15
running_loss_history = []
running_corrects_history = []
val_running_loss_history = []
val_running_corrects_history = []

for e in range(epochs):
    running_loss = 0.0
    running_corrects = 0.0
    val_running_loss = 0.0
    val_running_corrects = 0.0

    for inputs, labels in training_loader:
        inputs = inputs.view(inputs.shape[0], -1)     # flatten anneng vor 28x28@ darna 784
        outputs = model(inputs)                        # predict using the model class
        loss = criterion(outputs, labels)             # calculate the loss

        optimizer.zero_grad()                         # make the gradients zero so as to not accumulate
        loss.backward()                               # backward is to calculate the gradients
        optimizer.step()                              # update the weights using the step fucntion

        _, preds = torch.max(outputs, 1)              # torch max@ veradarcnuma tuple 1 in@ softmaxi score na isk 2 rd@
                                                      # en classi index@ vor maximumna, mer mot de tvera stacvuma ete 0 i softmax
                                                      # score na amenamec@ kveradarcni 0 index@ u heto sranov karanq loss hashveng
        running_loss += loss.item()                   # running loss@ avelacnumeng vor batch size i loss@ karenaq hashveng
        running_corrects += torch.sum(preds == labels.data)  # correctne te amen batchum nkarneric qanisna chisht gushakum
```

```python
else:
    with torch.no_grad():
        for val_inputs, val_labels in validation_loader:
            val_inputs = val_inputs.view(val_inputs.shape[0], -1)   # flattten
            val_outputs = model(val_inputs)                         # predict
            val_loss = criterion(val_outputs, val_labels)           # calculate loss

            _, val_preds = torch.max(val_outputs, 1)
            val_running_loss += val_loss.item()                                # add up validation loss
            val_running_corrects += torch.sum(val_preds == val_labels.data)    # calcualte the exact number of correct prediction

        epoch_loss = running_loss/len(training_loader)                # hashvumenq amen epoch ic heto loss@
        epoch_acc = running_corrects.float()/ len(training_loader) # hashvumenq te batch size 100 hatic vor tokosna chisht gushak
        running_loss_history.append(epoch_loss)                       # list i mech append enq anum heto vor karenanq plt ov nkarenq
        running_corrects_history.append(epoch_acc)                    # eli listi mech append aneng vor heto karenanq tpenq

        val_epoch_loss = val_running_loss/len(validation_loader) # stex hashvum enq amen batch i loss@
        val_epoch_acc = val_running_corrects.float()/ len(validation_loader) # hashvumenq accuracy n
        val_running_loss_history.append(val_epoch_loss)
        val_running_corrects_history.append(val_epoch_acc)
        print('epoch :', (e+1))                                      # e + 1 vor 0 i tex@ 1 i epochic sksi tpel sax
        print('training loss: {:.4f}, acc {:.4f} '.format(epoch_loss, epoch_acc.item()))
        print('validation loss: {:.4f}, validation acc {:.4f} '.format(val_epoch_loss, val_epoch_acc.item()))
```

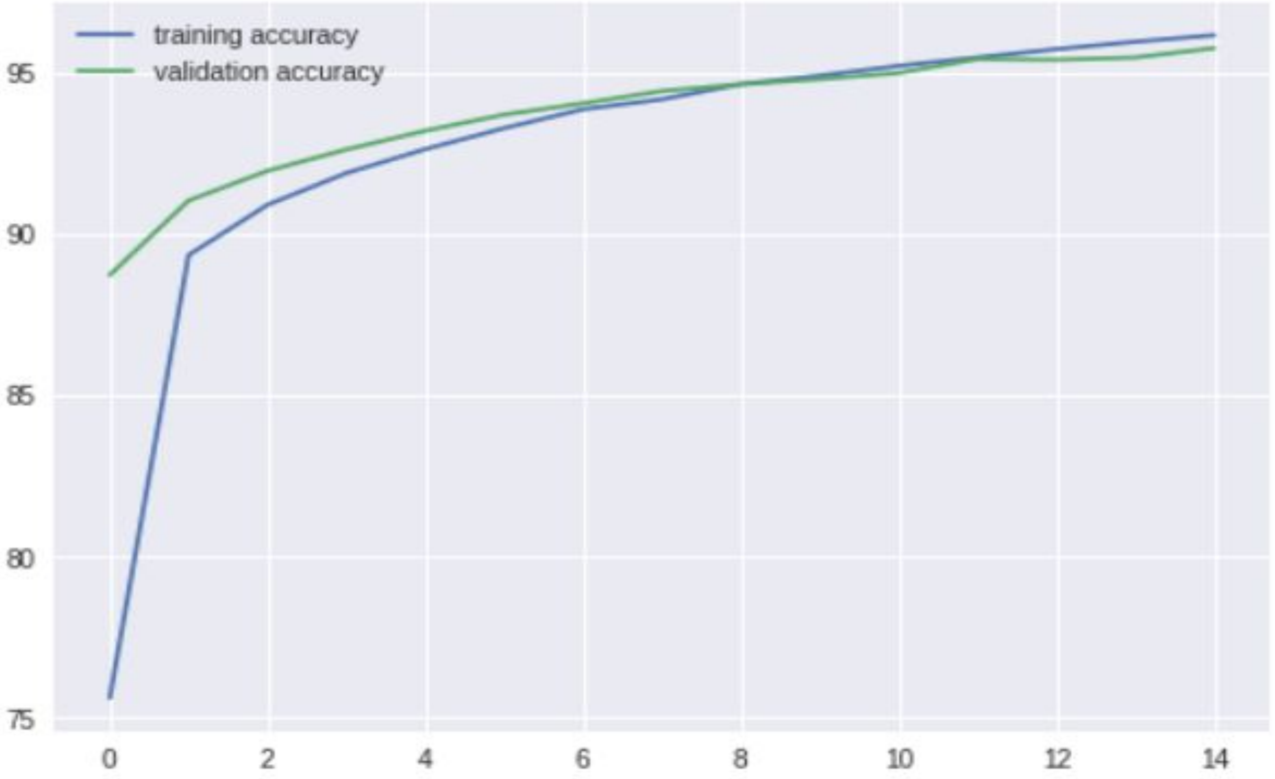
```
plt.plot(running_loss_history, label='training loss')
plt.plot(val_running_loss_history, label='validation loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb4140be80>
```



```
plt.plot(running_corrects_history, label='training accuracy')
plt.plot(val_running_corrects_history, label='validation accuracy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fcb456383c8>
```



```
dataiter = iter(validation_loader)
images, labels = dataiter.next()
images_ = images.view(images.shape[0], -1)
output = model(images_)
_, preds = torch.max(output, 1)

fig = plt.figure(figsize=(25, 4))

for idx in np.arange(20):
    ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
    plt.imshow(im_convert(images[idx]))
    ax.set_title("{} ({})".format(str(preds[idx].item()), str(labels[idx].item())), color=("green" if preds[idx]==labels[idx] els
```