Tutorial 1

Deep learning review

Some slides were based on lectures from the course “Intro to machine learning” by prof. Nir Rozenfeld.

Outline

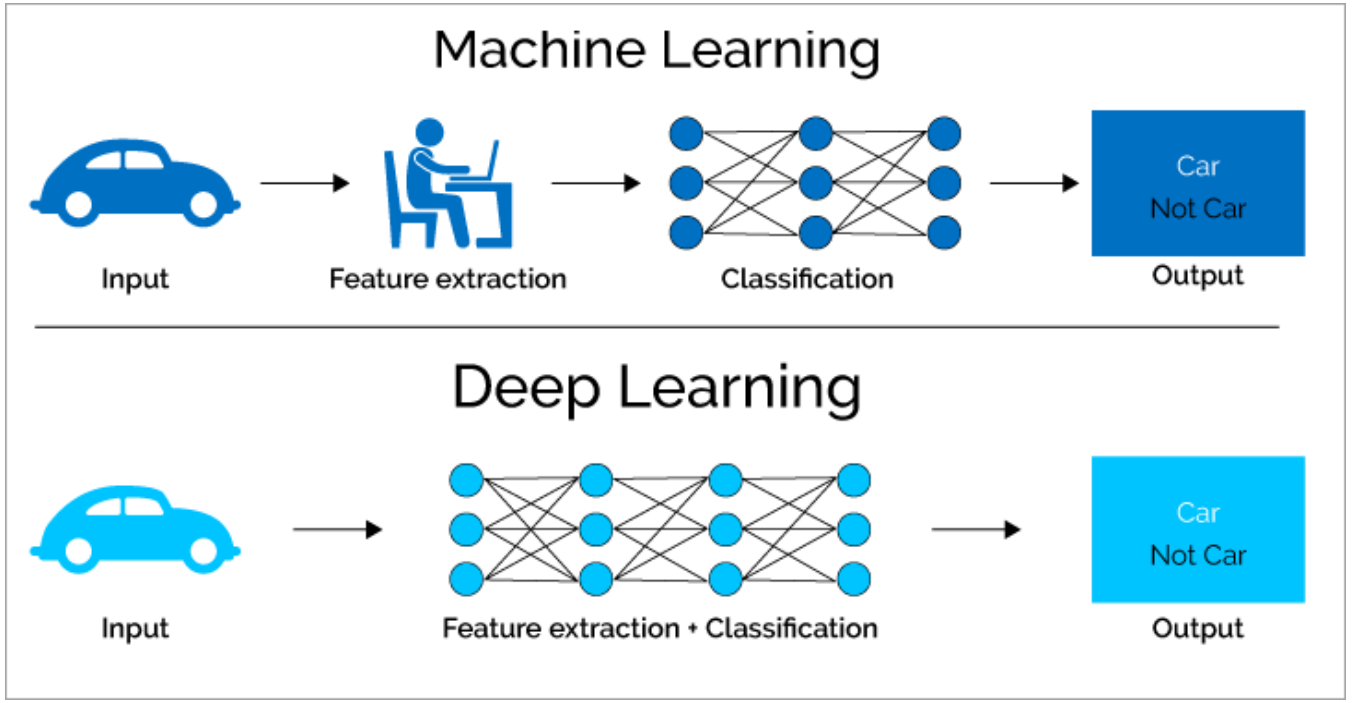
- Deep learning concepts
- PyTorch concepts
- Diagnosing loss functions

Deep learning

A class of machine learning models,

based on **stacked, differentiable** (linear/non linear) transformations,

that also **learn the representation** from data.



Tabular Input to DL models

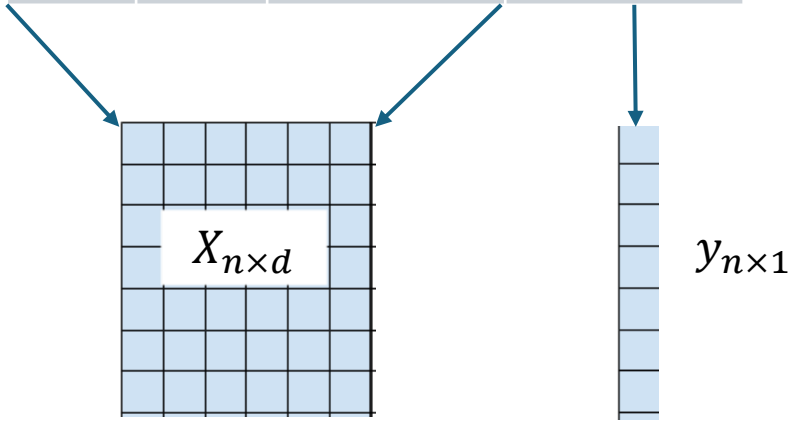
Deep learning models can handle numeric input.

How to handle categorical features?

- One-hot encoding – we will use this today
- Embedding using a language model
- Many other ways..

Eventually we get a numeric matrix.

Age	BMI	Smoking	Diabetes
23	25	current	0
67	31	former	1
50	29	never	0



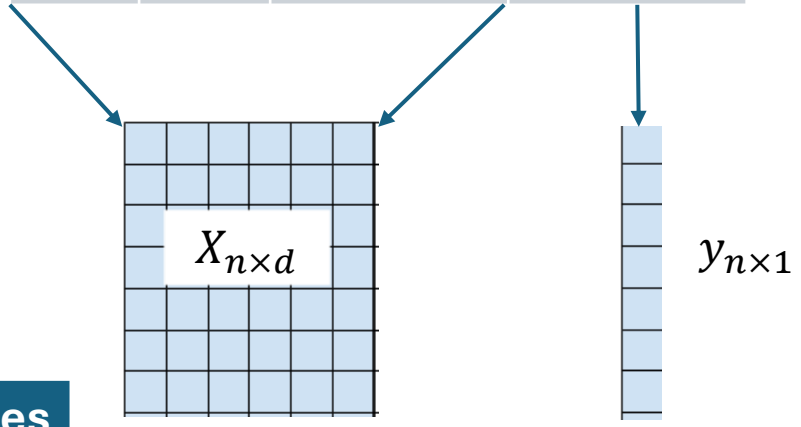
Tabular Input to DL models

```
import pandas as pd

df = pd.read_csv("../datasets/diabetes_prediction_dataset.csv", index_col=0)
df_encoded = pd.get_dummies(df, columns=['smoking_history'], dtype=int)

X = df_encoded.drop('diabetes', axis=1)
y = df_encoded['diabetes']
```

Age	BMI	Smoking	Diabetes
23	25	current	0
67	31	former	1
50	29	never	0



X

Age	BMI	Smoking_current	Smoking_former	Smoking_never
23	25	1	0	0
67	31	0	1	0
50	29	0	0	1

y

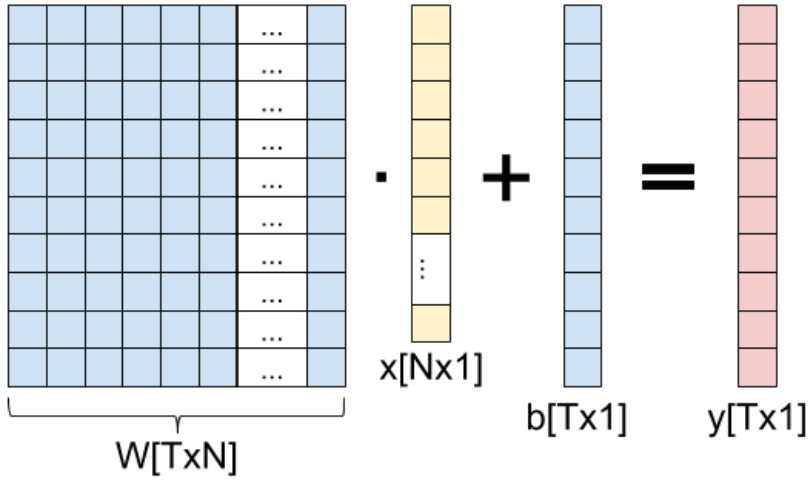
Diabetes
0
1
0

Linear layers

Fully connected

$$y = Wx + b$$

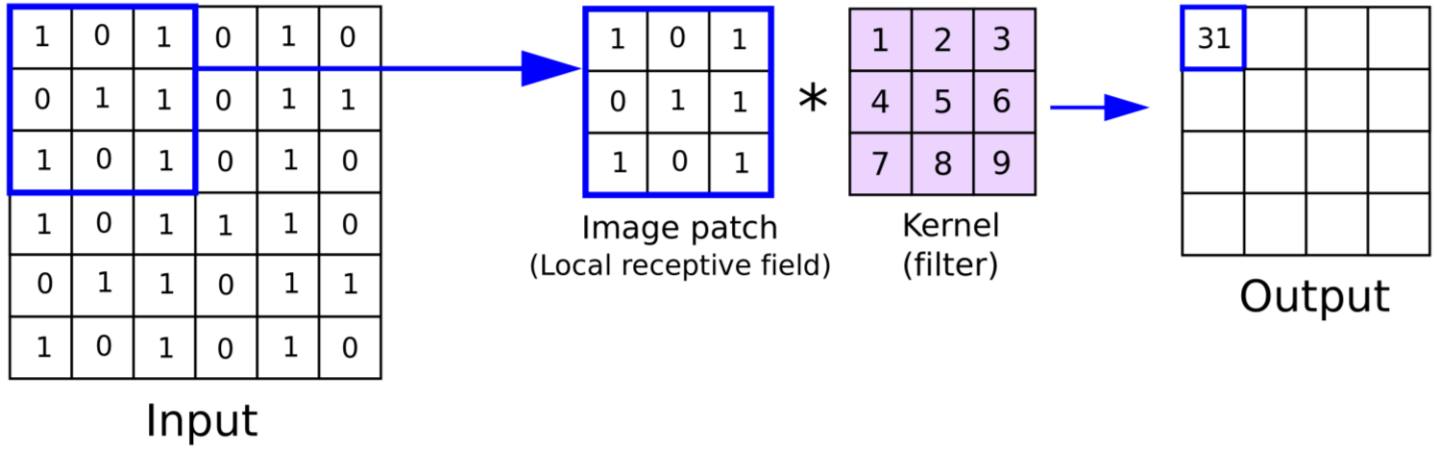
Learned weight matrix — input — bias



Convolution

$$y = Wx + b$$

Learned "kernel" — Local receptive field — bias

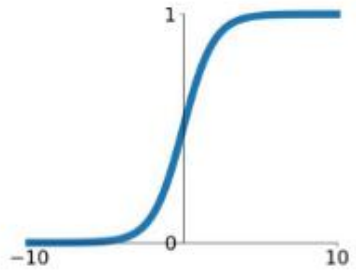


Non-linear layers - elementwise

Non-linear layers (or “activation functions”) add expressive power.

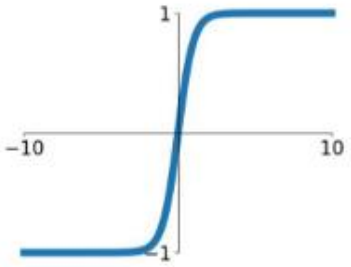
sigmoid:

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



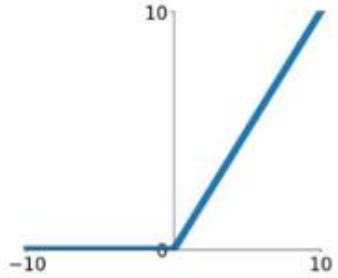
tanh:

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



ReLU:

$$\sigma(x) = \max\{0, x\}$$

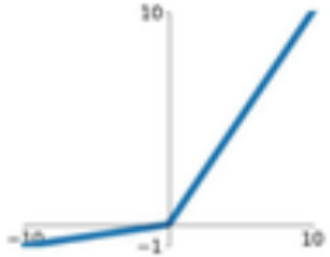


- These functions are applied on each element (separately).
- Used in hidden layers of the network.

To help with vanishing gradients:

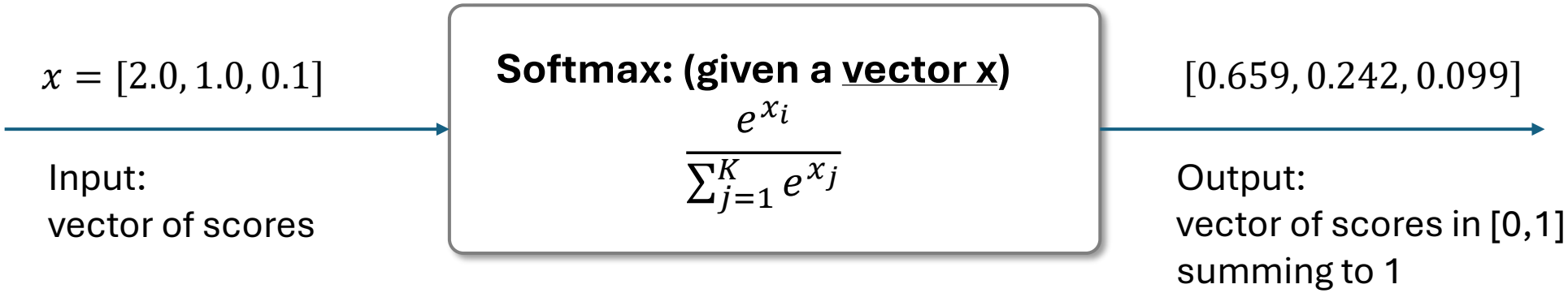
Leaky ReLU:

$$\sigma(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$



Non-linear layers - vectorwise

Non-linear layers (or “activation functions”) add expressive power.



- Softmax is applied on a vector of scores.
- The output is a vector of scores in [0,1], that sum to 1.
  - Acts as a probability distribution.
- Can be used in the output layer of the network (for classification).

Loss

- A measure of the quality of the model.
- The optimization (learning process) aims to **minimize the loss**.
- In supervised learning, the loss is usually a measure of **distance** between the **model output** and a **ground truth label**.

For classification
difference between distributions

Cross Entropy:

$$H(p,q) = -\sum_x p(x) * \log q(x)$$

p – the true distribution (1-hot)
q – the model output

For regression
difference between numeric label and prediction

L1:

$$\frac{1}{n}\sum_i |y_i - \hat{y}_i|$$

L2 (Mean Square Error):

$$\frac{1}{n}\sum_i (y_i - \hat{y}_i)^2$$

Outline

- Deep learning concepts
- PyTorch concepts
- Diagnosing loss functions

PyTorch Basics: Defining the Model

- **PyTorch** - An open-source deep learning framework for building and training neural networks.
- Networks are defined using a dynamic computation graph.

```python
class Classifier(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(input_dim, 32)
        self.layer2 = nn.Linear(32, 16)
        self.output = nn.Linear(16, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        return self.output(x)
```

In `__init__`: define the layers (building blocks)

Inherit from `nn.Module`

Backward is implemented automatically, using the partial derivatives of each component.

Implement **`forward`** to define the computation graph, connecting the layers

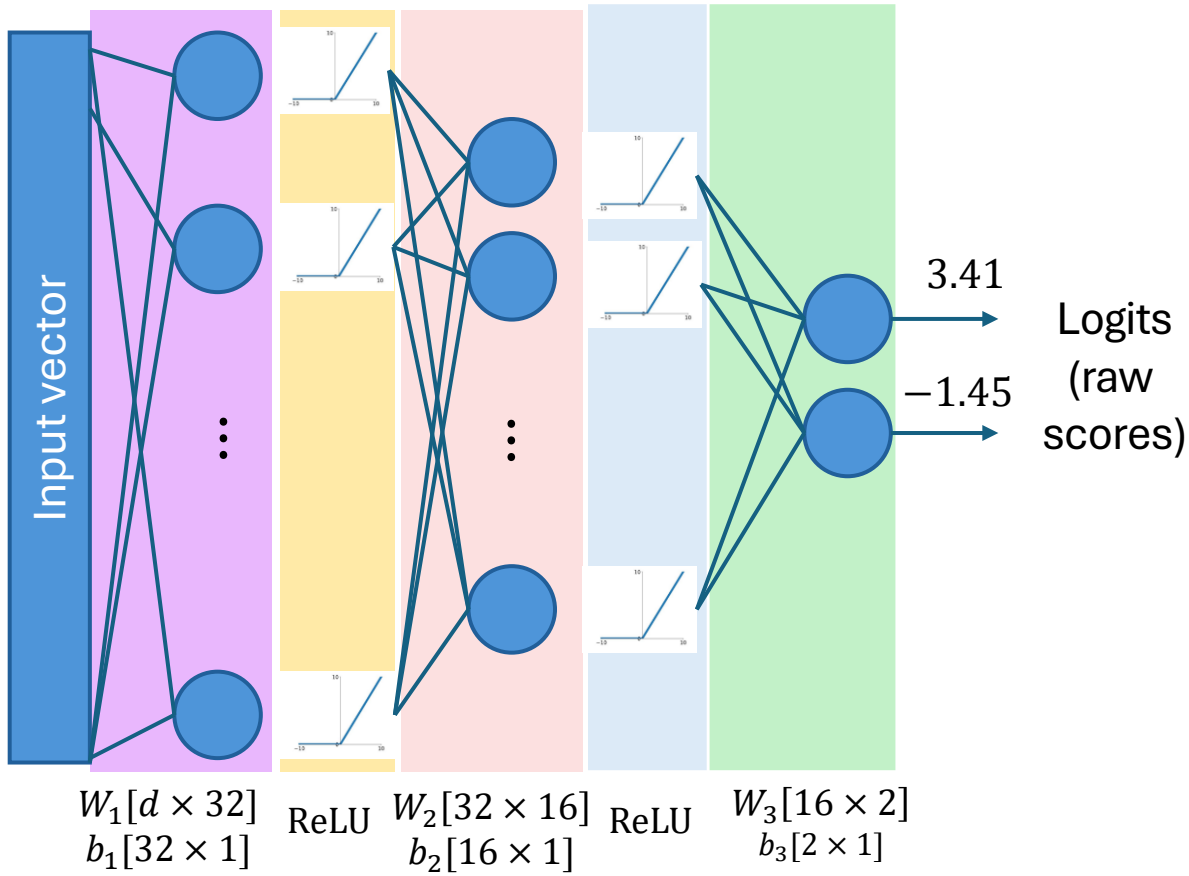
What does this model look like?

PyTorch Basics: Defining the Model

What does this model look like?

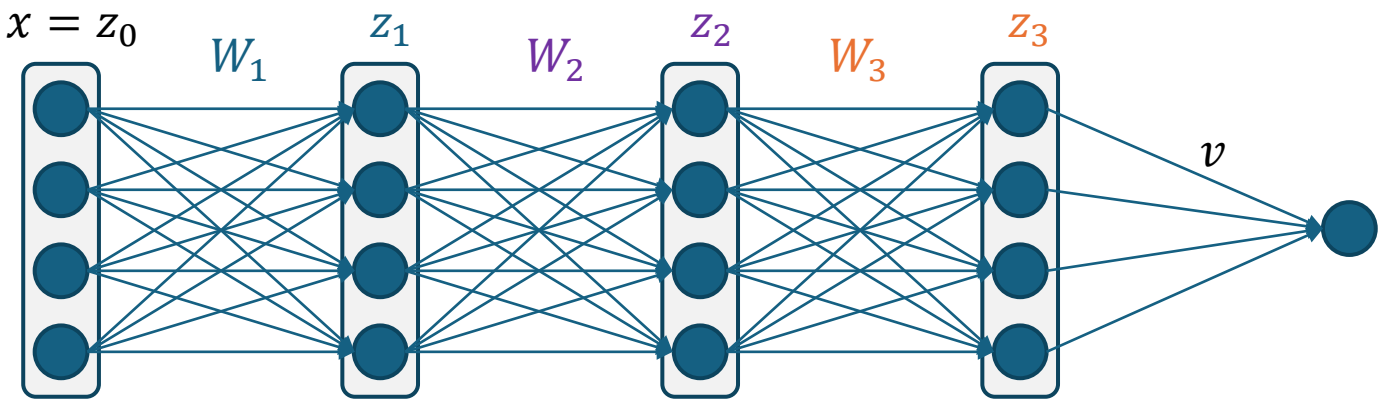
```
class Classifier(nn.Module):
    def __init__(self, input_dim, num_classes):
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(input_dim, 32)
        self.layer2 = nn.Linear(32, 16)
        self.output = nn.Linear(16, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        return self.output(x)
```



Which layers are trained?
(What is changed during training?)

Backpropagation - Reminder



$$\hat{y} = h(x) = \sigma(v^{\top}\sigma(W_{T-1}\sigma(W_2\sigma(W_1 x))))$$

$$\hat{y} \longrightarrow Loss = f(\hat{y}, y)$$

- We want to minimize the loss using **gradient descent**: $W_{new} = W_{old} - \mathrm{lr} \cdot \frac{\partial\, Loss}{\partial W}$
- Need to calculate $\frac{\partial\, Loss}{\partial W_i}$ for every weight.
- The relationship between W_i and the $Loss$ is complicated (nested).
- The **Chain Rule** is used: traverse the graph from the loss **back** to the inputs, to calculate the partial derivatives along the way.

Example: $\frac{\partial\, Loss}{\partial W_2} = \frac{\partial\, Loss}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$

PyTorch Basics: Tensors and Autograd

- Tensors are the atomic unit in PyTorch.
- Multi-dimensional arrays (like NumPy arrays), which **track their history**, to enable automatic differentiation (using the chain rule).
- Can be saved on the CPU or the GPU.

```
x = torch.tensor([1.0, 1.5, 3], requires_grad=True)
```

- Creates `x.grad` buffer, of the same shape as **x**.
  - This is where partial derivatives are saved when we later call `.backward()`.
- `requires_grad` is "contagious" - true for any tensor based on x.
  - If `y=f(x)`, the attribute `y.grad_fn` will be used to track the derivative path.
- In `nn.Module`, `requires_grad=True` by default for learned parameters.
  - Like W and b for linear layers.

PyTorch: Training Loop

```
loss_func = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.01)

train_ds = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_ds,
                          batch_size=128, shuffle=True)
for epoch in range(30):
    model.train()
    for batch_X, batch_y in train_loader:
        # 1. Forward
        outputs = model(batch_X)
        loss = loss_func(outputs, y_train)

        # 2. Backward (auto grad)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Controls how the weights are updated

Handle batching

Enable training behavior*

In each batch: the forward pass computes the loss

Discard previous gradient info

Compute partial derivatives $\frac{\partial \text{ loss}}{\partial w}$ for each parameter w, by traversing the graph from the loss backwards

Update weights

PyTorch: Evaluation

```
model.eval()
with torch.no_grad():
    logits = model(X)
    loss = loss_func(logits, y).item()

    probs = torch.softmax(logits, dim=1)[:, 1]

    _, predicted = torch.max(logits, dim=1)
    y_pred = predicted.cpu().numpy()
    y_true = y.cpu().numpy()

    # Metrics
    accuracy = (y_pred == y_true).mean() * 100
    auc = roc_auc_score(y_true, probs)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
```

Disable training behavior*

Forward pass to get logits and compute loss

Get binary predictions, copy to cpu if necessary, convert to numpy arrays for metrics computation

Don't update gradients

Apply softmax to get probabilities, take the predictions for the positive class

Differences between these metrics – next slide

Classification Evaluation Metrics - Reminder

- **Accuracy** - % of correct predictions

$$\frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision** - % of positive predictions that were true

$$\frac{TP}{TP + FP}$$

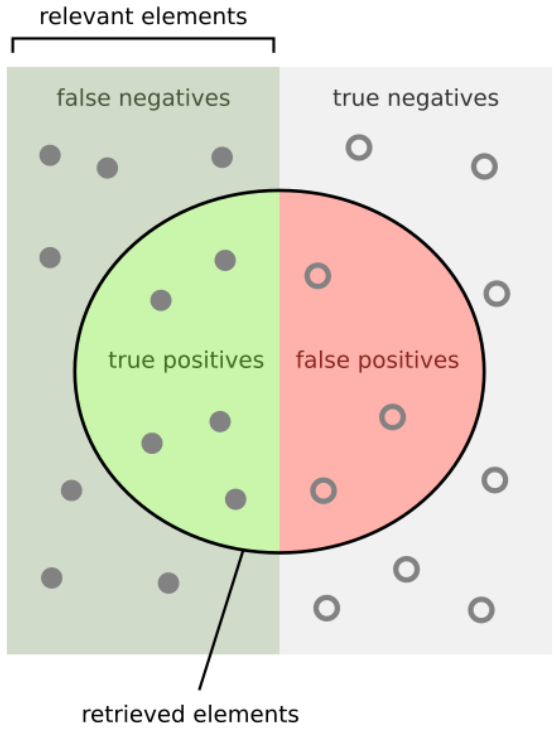
- **Recall** - % of true positive samples that were predicted true

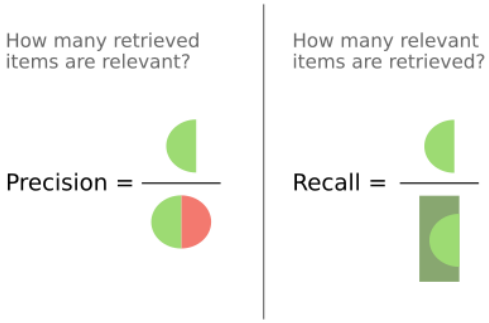
$$\frac{TP}{TP + FN}$$

- **ROC AUC** (Area under the ROC curve) - The model's ability to separate the classes.

$$P\left(h(x_{neg}) < h(x_{pos})\right)$$

These are just a few examples – there are many metrics.

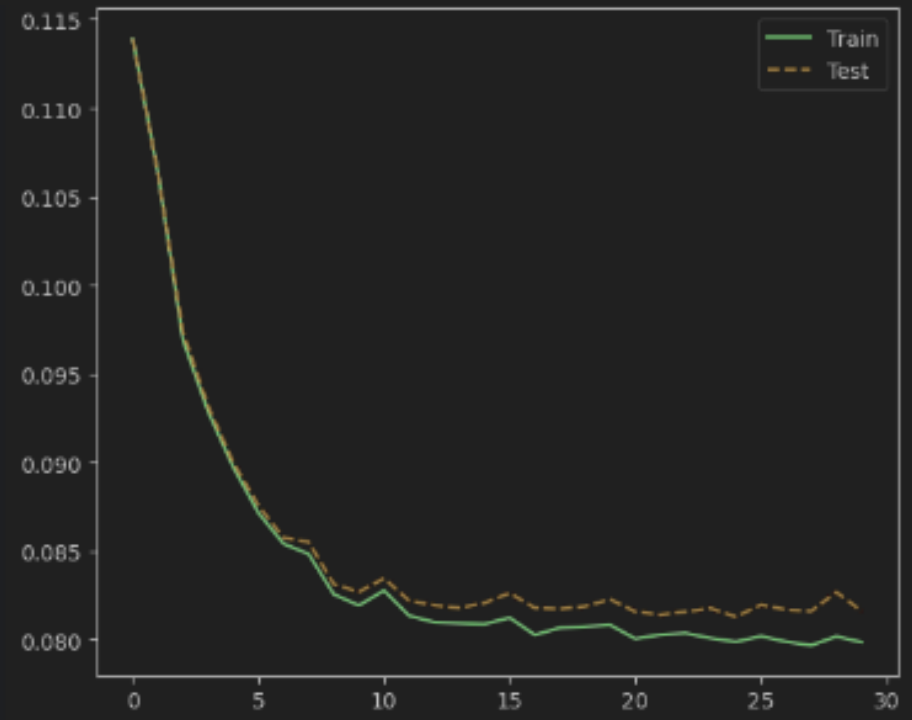




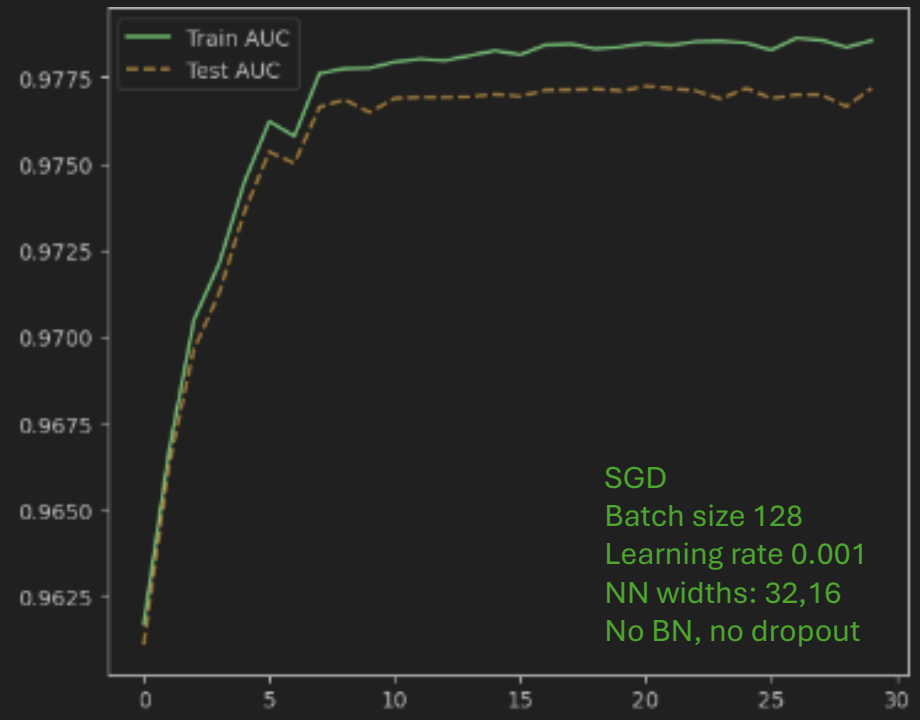
Metrics over Epochs

Lower is better

Cross Entropy loss



ROC AUC



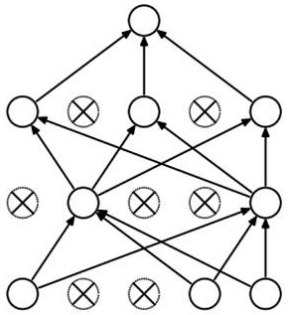
Higher is better

Loss is reducing nicely ☺

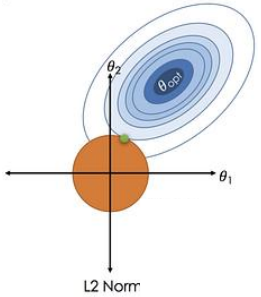
Test performance is not as good as train performance ☹

Regularization

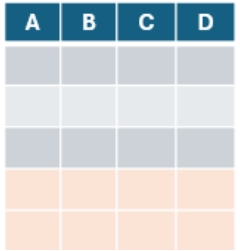
Possible ways to prevent overfitting and stabilize the training:



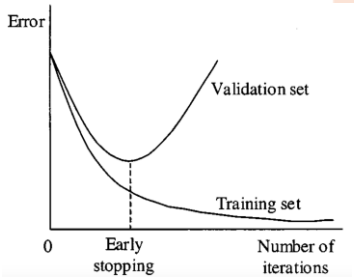
- **Dropout:** Randomly disable a fraction of the neurons during training.
  - If a neuron cannot rely on specific neighbors, it is forced to learn more robust features.



- **L2/L1 regularization:** Penalize large weights.
  - For example, by adding a term to the loss function: $\lambda||W||_2^2$



- **Data augmentation:** Artificially increase dataset size.
  - For example, by small changes on the original data or synthetic data generation.



- **Early stopping:** Stop training when validation error increases.

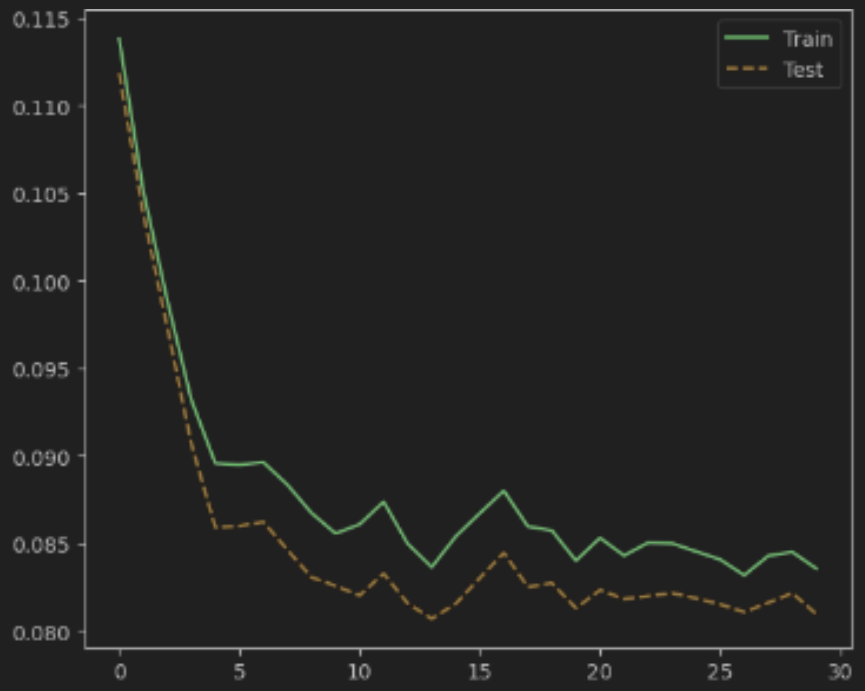
Normalization

- In practice, training can be slow to converge or unstable.
- To handle this, we can normalize the input/vectors in latent space.
- **Batch Normalization**: normalizes **each feature** independently across the entire batch.
  - Example: compute the average *Age (*or *BMI,* or *blood glucose)* and scale it, over a batch of 128 people.
  - Allows higher learning rates.
  - But depends on batch size (e.g., average of 2 is noisy).
- There is also **Layer Normalization -**
  - Computed over features of a single sample
  - Useful in RNNs and transformers
  - We won't go into it now.

With BN and Dropout

Lower is better

Cross Entropy loss



Epochs

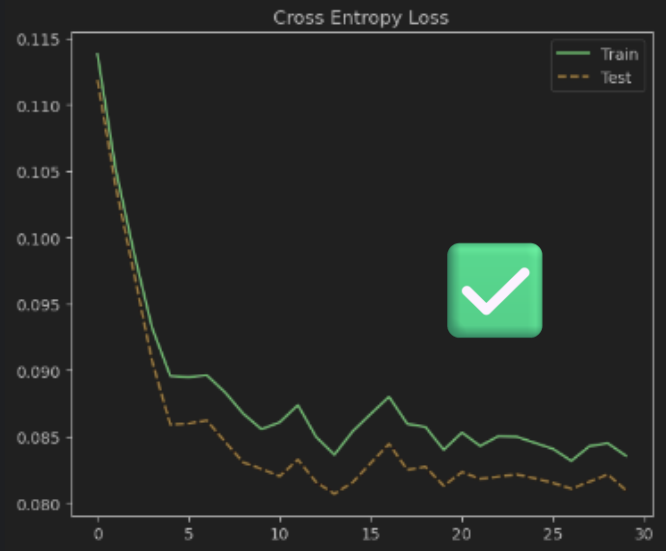
ROC AUC

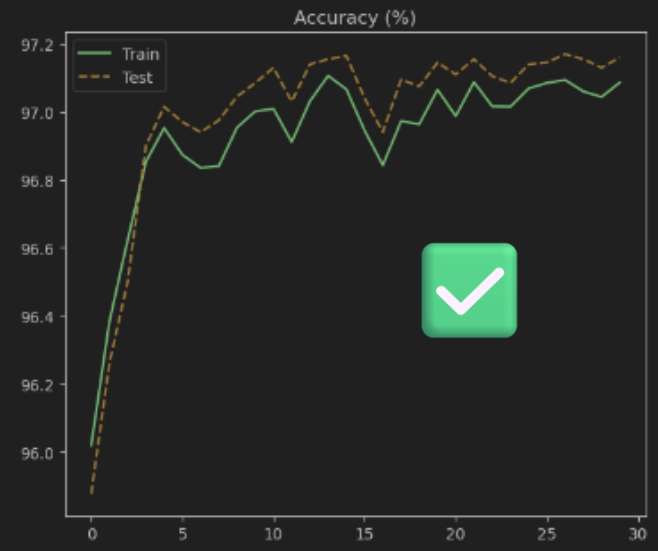
Epochs

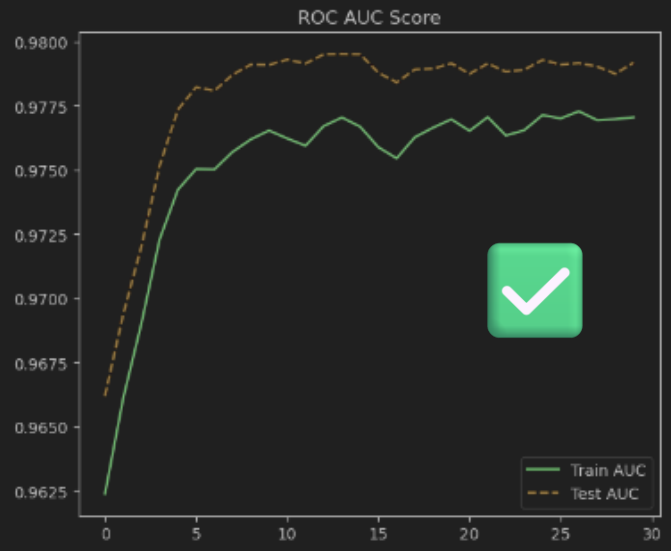
Higher is better

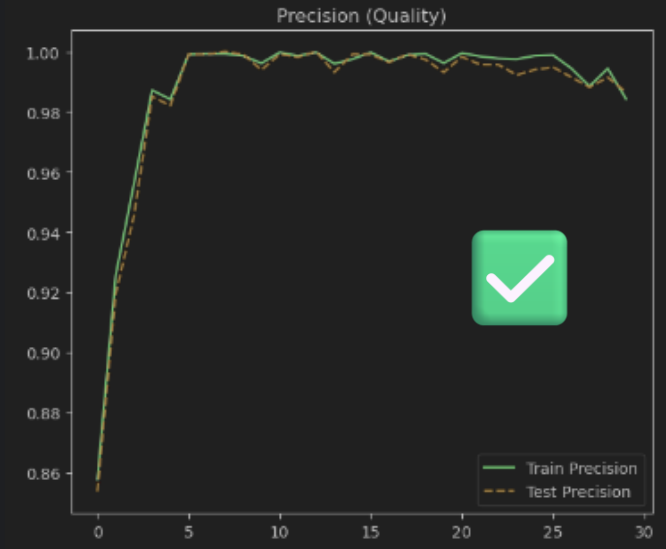
Looks like test performance is even better than train performance.
But are we seeing the full picture?

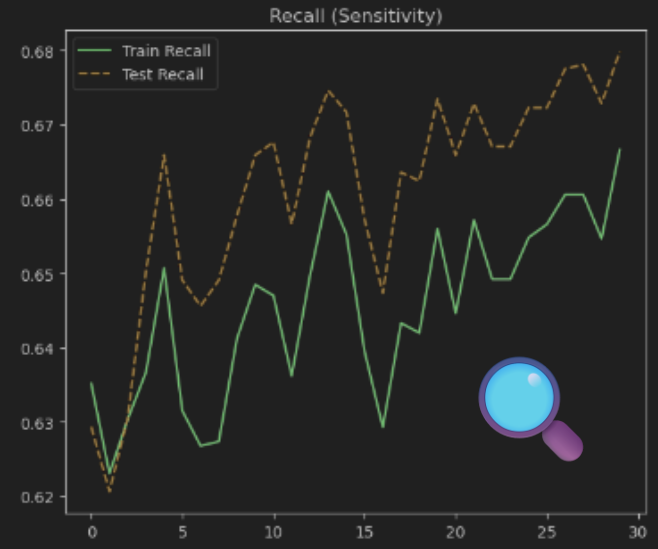
More Evaluation Metrics











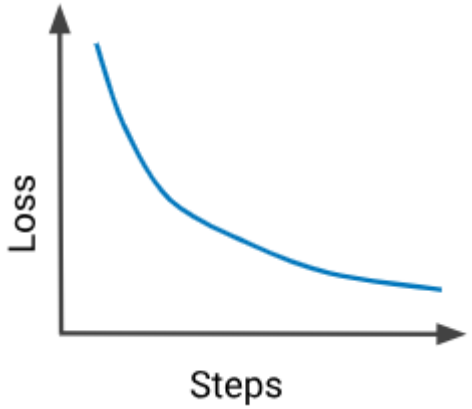
Loss, accuracy, AUC, even precision are okay
⇒ Positive predictions are indeed diabetes patients
But recall is not great
⇒ 32% of diabetes patients are predicted wrong
Why?
The data is very **imbalanced**: only 8.5% positive.
Possible solutions: data augmentation, custom loss function, many more options.

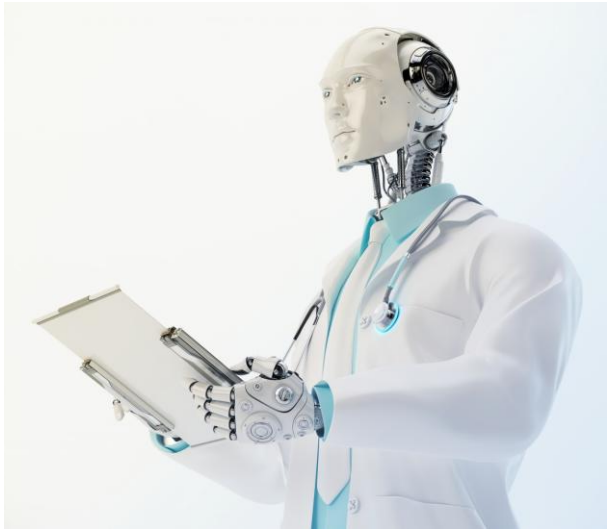
Outline

- Deep learning concepts
- PyTorch concepts
- Diagnosing loss functions

The Loss Doctor

Doctor, what is wrong with my loss?



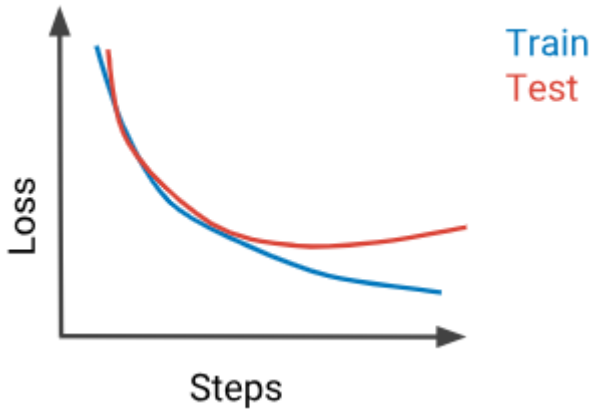


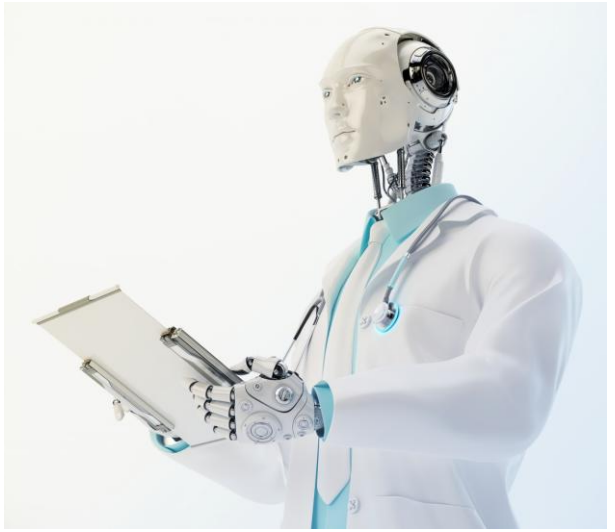
Answer: nothing, it’s perfect

Source: https://developers.google.com/machine-learning/crash-course/overfitting/interpreting-loss-curves

The Loss Doctor

Doctor, what is wrong with my loss?

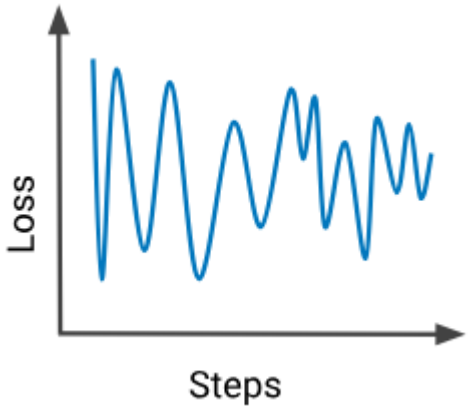


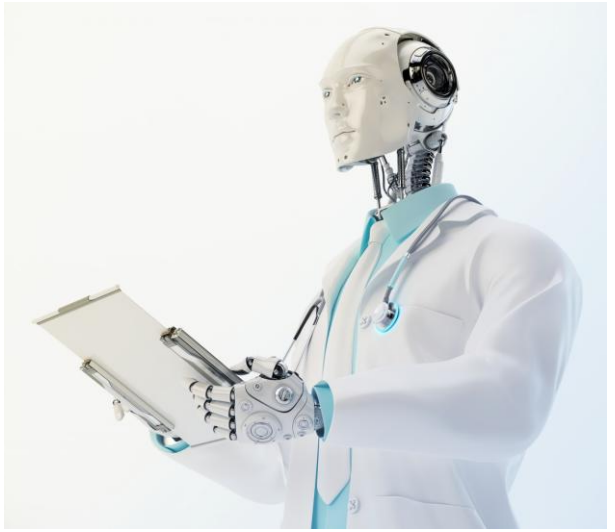


Possible Answers:
The model is overfitting on the training set.
Try (1) a simpler model, or (2) more regularization,
or (3) check if the train and test are statistically equivalent

The Loss Doctor

Doctor, what is wrong with my loss?

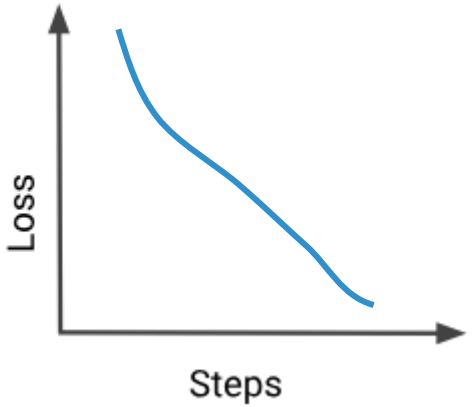


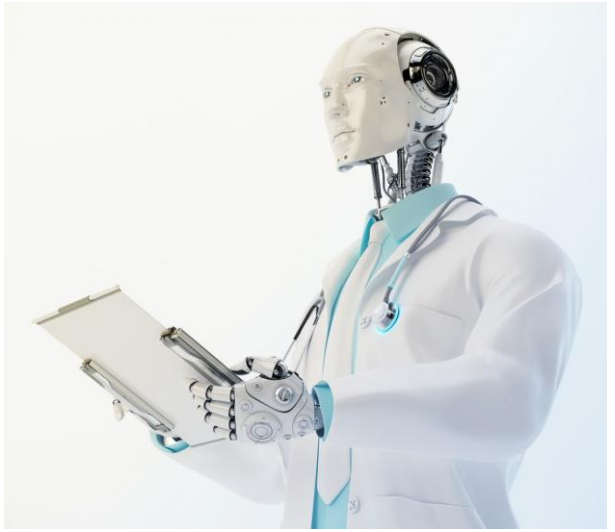


Possible answers:
(1) learning rate too large, or
(2) Some bad examples in the training set

The Loss Doctor

Doctor, what is wrong with my loss?





Training is not finished – try giving it more epochs

Summary

- We discussed the building blocks of deep neural networks
- Every part we discussed has many options and alternatives:
  - layers, non-linear activations, loss functions, optimizers, regularization, data augmentation, normalization..
- We covered these concepts so that you know what to search when you are building your own models.
- Recommendation: start simple, complicate if needed.