Is there also a backward pass?

- Backward pass, or backpropagation is used to update weights and biases during training
- In the training loop, we:

  1. Propagate data forward
  2. Compare outputs to true values (ground-truth)
  3. Backpropagate to update model weights and biases
  4. Repeat until weights and biases are tuned to produce useful outputs

```
import torch
import torch.nn as nn
input_data = torch.tensor([[..., ...], [..., ...]])
```

Binary classification: forward pass

```
model = nn.Sequential(
        nn.Linear(6,4),
        nn.Linear(4,1),
        nn.Sigmoid(),
        )

output = model(input_data)

print(output)
>> tensor([[0.5188],
           [0.3761], ...
```

Outputs:

- probabilities between 0 and 1
- one value for each sample (row) in data

Multi-class classification: forward pass

```
n_classes = 3
model = nn.Sequential(
        nn.Linear(6,4),
        nn.Linear(4, n_classes),
        nn.Softmax(dim=-1)
        )

output = model(input_data)

print(output.shape)
>> torch.Size([5,3])  # 3 classes
```

Outputs:

- The output dimension is 5x3
- Each row sums to 1
- Value with highest probability is assigned predicted label in each row

Regression: forward pass

```
model = nn.Sequential(
        nn.Linear(6,4),
        nn.Linear(4,1)
        )

output = model(input_data)

print(output)
>> tensor([[0.3818],
           [0.0712],
           ... ])
```

Output

- 5x1
- 5 continuous values, one for each row.

Loss function:

- The loss function tells us how good our model performs during training
- Takes in model prediction $\hat{y}$ and ground truth y
- Outputs a float
- Our goal is to minimize the loss-function!!!

$$loss = F(y, \hat{y})$$

- $\underline{y}$ is a single **integer** (class label)
  - e.g. $y = 0$ when y is a mammal
- $\underline{\hat{y}}$ is a **tensor** (output of softmax)
  - If N is the number of classes, e.g. $N = 3$
  - $\hat{y}$ is a tensor with N dimensions

    ($\hat{y} = [0.57492, 0.034961, 0.156647]$)

→ **Question:** How do we compare an integer to a tensor to evaluate model performance?

One-hot encoding concepts:

Transforming true label to tensor of 0 and 1.

For example: truth: $y=0$

Classes: $N=3$

0	1	2	classes
1	0	0	one-hot encoding

$\rightarrow$ $y=0$ is $[1,0,0]$

Transforming labels with one-hot encoding

```
import torch.nn.functional as F
F.one_hot(torch.tensor(0), num_classes=3)
  >> tensor([1,0,0])
F.one_hot(torch.tensor(1), num_classes=3)
>> tensor([0,1,0])
```

The most used loss function for classification problems: <u>Cross-entropy Loss</u>

Cross entropy loss in PyTorch:

```
from torch.nn import CrossEntropyLoss

scores = tensor([[-0.1211, 0.1059]])  # ŷ
one_hot_target = tensor([[1,0]])

criterion = CrossEntropyLoss()
criterion(scores.double(), one_hot_target.double())

>> tensor(0.8131, dtype=torch.float64)     # loss value
```

Summary:

- scores ⇒ model predictions **before** the final softmax function
- one_hot_target ⇒ one hot encoded ground truth label and outputs
- loss ⇒ a single float

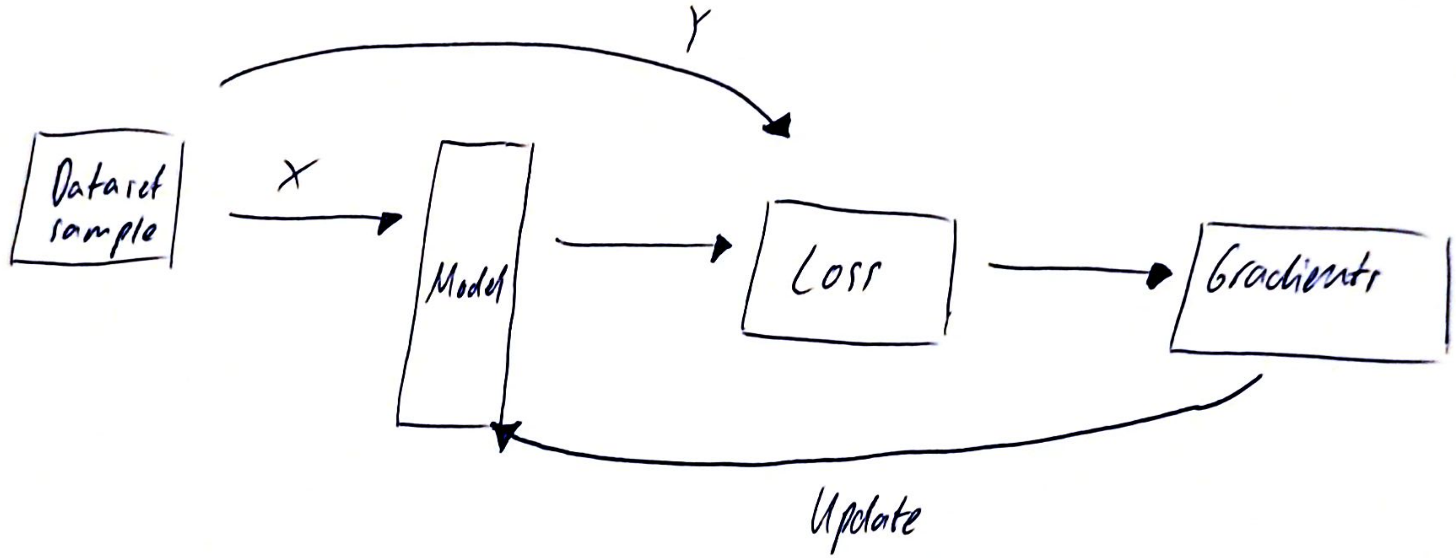
⇒ GOAL: MINIMIZE LOSS FUNCTION!

Minimizing the loss

High loss : model prediction is wrong

Low loss : model prediction is correct

Model training: Updating a model's parameter to minimize the loss



- We take a dataset with features X and ground truth y. We run a foward pass using X and calculate loss by comparing model output; $\hat{y}$, with y.

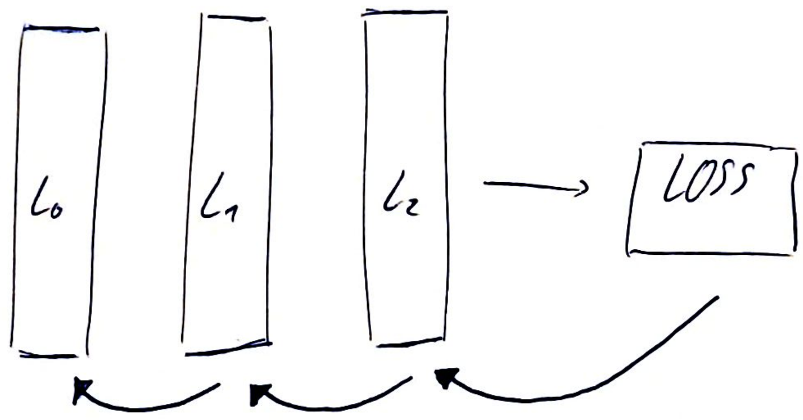
- We compute gradients of the loss function and use them to update the model parameters with backpropagation, so that weights are no longer random and biases are useful.

Backpropagation concepts

- Consider a network made of 3 Layers : L_0, L_1 and L_2

↳ We calculate local gradients for L_0, L_1 and L_2 using <u>backpropagation</u>

We calculate loss gradients with respect to L_2, then use L_2 gradients to calculate L_1 gradients, and so on...



Backpropagation

Backpropagation in PyTorch

```
# Create the model and run a forward pass
model = nn.Sequential(nn.Layer(16,8),
                      nn.Layer(8,4),
                      nn.Layer(4,2))

predictions = model(sample)

# Calculate the loss and compute the gradients
criterion = CrossEntropyLoss()
loss      = criterion(prediction, target)
loss.backward()

# Access each layer's gradients
model[0].weight.grad,  model[0].bias.grad
model[1].weight.grad,  model[1].bias.grad
model[2].weight.grad,  model[2].bias.grad
```

→ Each layer has a weight, a bias and the corresponding gradients

Updating model parameters

Update the weights by subtracting local gradients scaled by the learning rate

```
# Learning rate is typically small
lr = 0.001

# Update the weights
weight = model[0].weight
weight-grad = model[0].weight.grad
weight = weight - lr * weight-grad

# Update the biases
biases = model[0].biases
bias_grad = model[0].bias.grad
bias = bias - lr * bias_grad
```

Convex and non-convex function:

convex:  non-convex: 

When minimizing loss-function, our goal is to find the global minimum of the non-convex function.

Loss functions used in deep learning are non-convex!

→ To find global minima of non-convex functions, we use a mechanism called "gradient descent"

PyTorch used optimizer: Most common SGD: , SGD = Stochastic gradient descent

```
import torch.optim as optim
#Create the optimizer
optimizer = optim.SGD(model.parameters(), lr = 0.001)
optimizer.step()      #updating parameter
```

Training a neural network – Summary:

1. Create a model
2. Choose a loss function
3. Create a dataset
4. Define an optimizer
5. Run a training loop, where for each sample of the dataset, we repeat:
   - Calculate loss (forward pass)
   - Calculating local gradients
   - Updating model parameters

```
# Create the dataset and the dataloader
dataset = TensorDataset(torch.tensor(features).float(), torch.tensor(target).float())
dataloader = DataLoader(dataset, batch_size = 4, shuffle = True)

# Create the model
model = nn.Sequential(nn.Linear(4,2),
                      nn.Linear(2,1))

# Create the loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr = 0.001)
```