Section 5: Feed Forward NN

Lecture 23 artificial Neural networks Intro

ANN → Artificial Neural Net

→ Logistic Regression is a Single Neuron

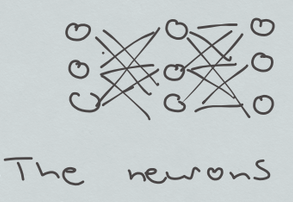
Section Outline

1. Model architecture
2. Geometrics
3. activation function
4. Multiclass classification
5. Image data.

Lecture 25: Forward propagation

Every neuron can focus on different features

→ Multiple layers of neurons



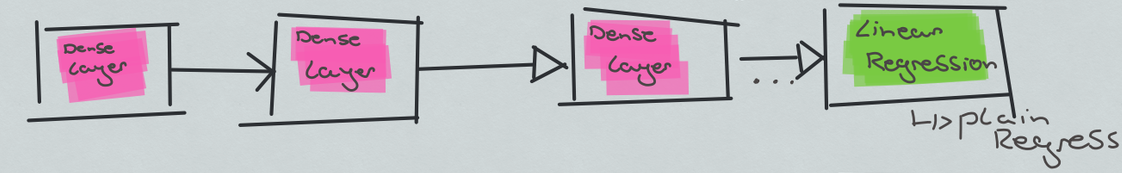
→ The neurons are vectorized to make the calculation more efficient.

→ Final layers every input of a layer is the output of the previous layer.

In classification you want a sigmoid at the end (0,1). In Regression you don't want a single sigmoid at the end.

Each neural layer is a feature transformation

Note (Dense means fully connected)



Deep learning (Lots of layers) is able to learn hierarchical and very complex data.
↳ Break down in smaller problems

Example Facial Recognition
Layer 1 = Lines
Layer 2 = Parts of face
Layer 3 = Full Face.

Lecture 25: the geometrical Picture

The Neural Model is linear
Multiple Neurons allow for curved lines allowing for more complex solving of data.

W & B have no input that can give a curve

→ You also use feature engineering to transpose to linearity and make a problem solvable.

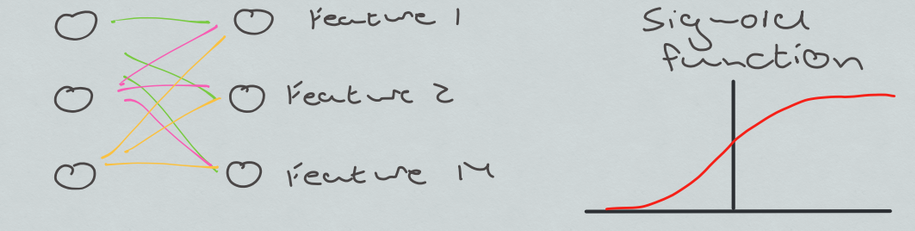
If you have multiple inputs and you feature engineer you have interaction terms which grow exponentially.

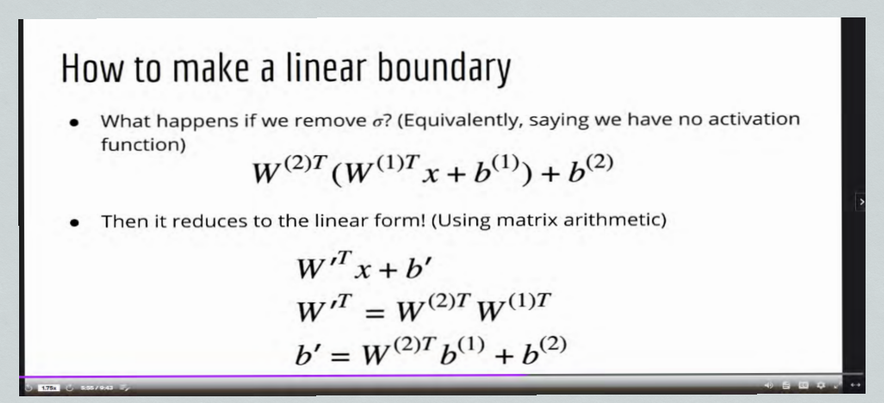
3 Inputs → 9 variables.

x_1, x_2, x_3
x_1^2, x_2^2, x_3^2
x_1x_2, x_2x_3, x_3x_1

Neuron

each neuron computes a different non linear feature → Because of sigmoid function





How to make a linear boundary

- What happens if we remove σ? (Equivalently, saying we have no activation function)

$$W^{(2)T}(W^{(1)T}x+b^{(1)})+b^{(2)}$$

- Then it reduces to the linear form! (Using matrix arithmetic)

$$W'^{T}x+b'$$
$$W'^{T}=W^{(2)T}W^{(1)T}$$
$$b'=W^{(2)T}b^{(1)}+b^{(2)}$$

so without the sigmoid its linear and the complexity dissapears

Automatic Feature engineering

Because W, B are Randomized and found iteratively with gradient descent you don't have to manually select features and engineer them

eliminates need for domain knowledge

playground.Tensorflow.com

The feature selection is feature engineering

Neural Network doesn't need it but it should be noted that you need a complex network and lots of epoch

Lecture 26 activation functions

you want to standardize input -1, 0, 1 because the layer output is the next layer input. → uniformity

Sigmoid does 0, 0.5, 1
tanh does -1, 0, 1

every layer wants standardized inputs so tanh has a better output.

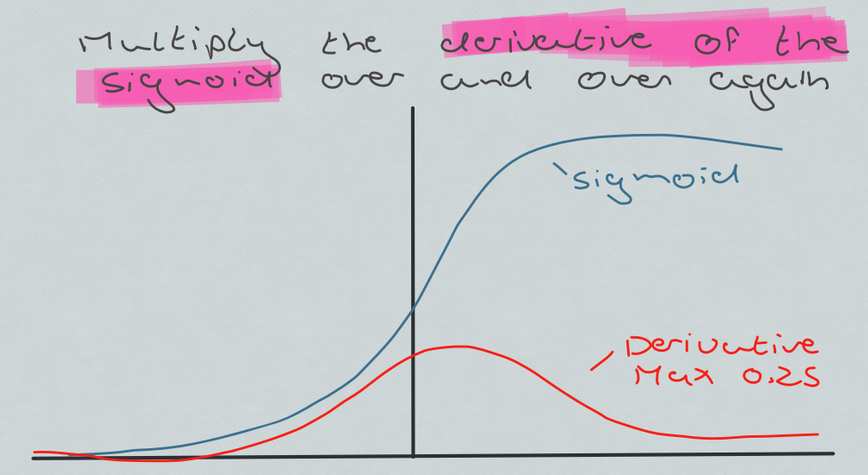
Cautionary tale

Both are still problematic

the vanishing gradient problem

↳ the deeper a network the more terms have to be Multiplied due to the chain Rule

Multiply the derivative of the sigmoid over and over again



Multipling small numbers by going back further in a neural network means the gradient becomes smaller and smaller

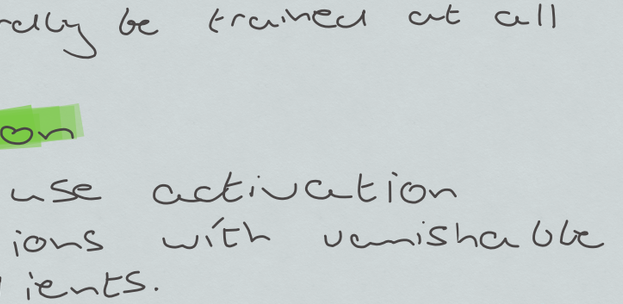
↳ this means the layer close to the input would hardly be trained at all

Solution

Don't use activation functions with vanishable gradients.

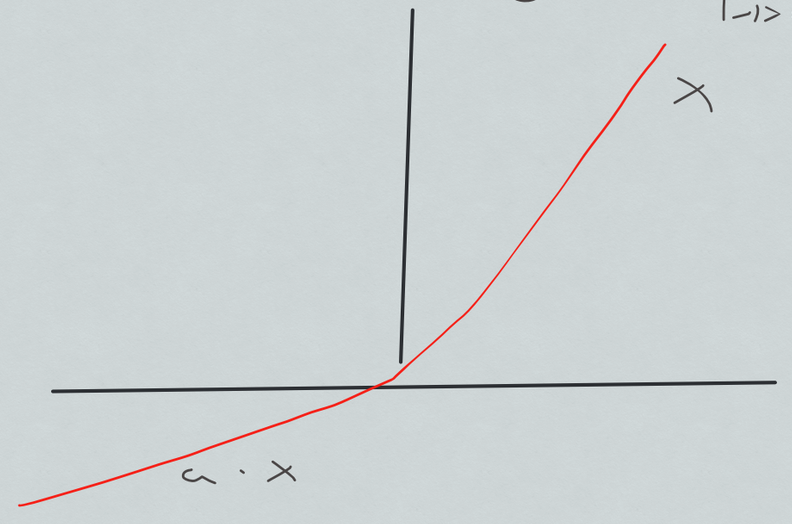
Like ReLU

↳ = Max(0,z)

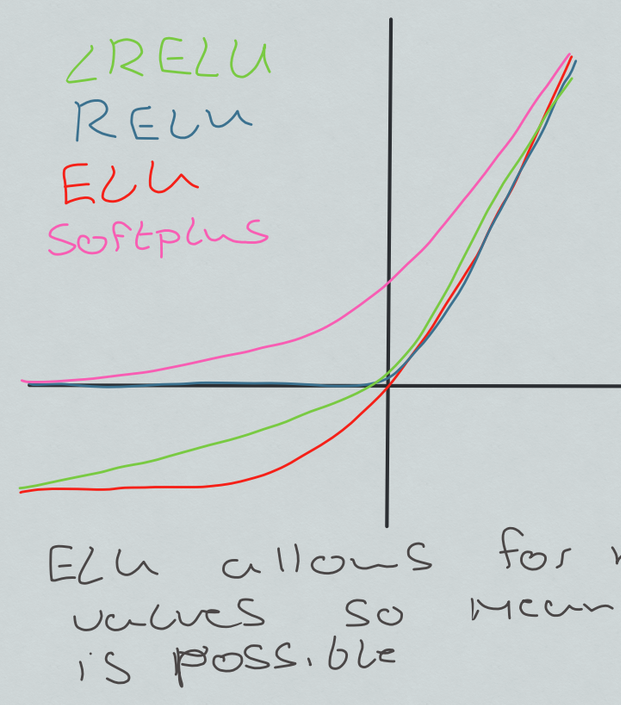


creates dead neuron problem but in application this doesn't matter

↳ solved by LReLU
↳ leaky



↳ and ELU
exponential linear unit



ELU allows for negative values so mean of 0 is possible

ELU + Softplus have vanishing gradients on the left but its fine since ReLU works.

RELU is a Reasonable default
test all possible RELU, ELU, softplus

Other functions: E1RU, E2RU, E3RU | O1RU, O2RU, O3RU

Lecture 27 Multiclass Classification

If you have k possible outcomes you need k output Nodes

every output has a probability
→ all probabilities must sum to 1.
Softmax function does this for you

Softmax in pytorch

```
nn.sequential(
    nn.Linear(D,M)
    nn.ReLU(),
    nn.Linear(M,k)
    nn.softmax()
)
```

Softmax is a activation function but not for hidden layers

→ Don't use this ⚠
Because exponentiating thing is Bad

Do it like so

```
model = Linear(D,k)
criterion = nn.crossentropyloss()
```

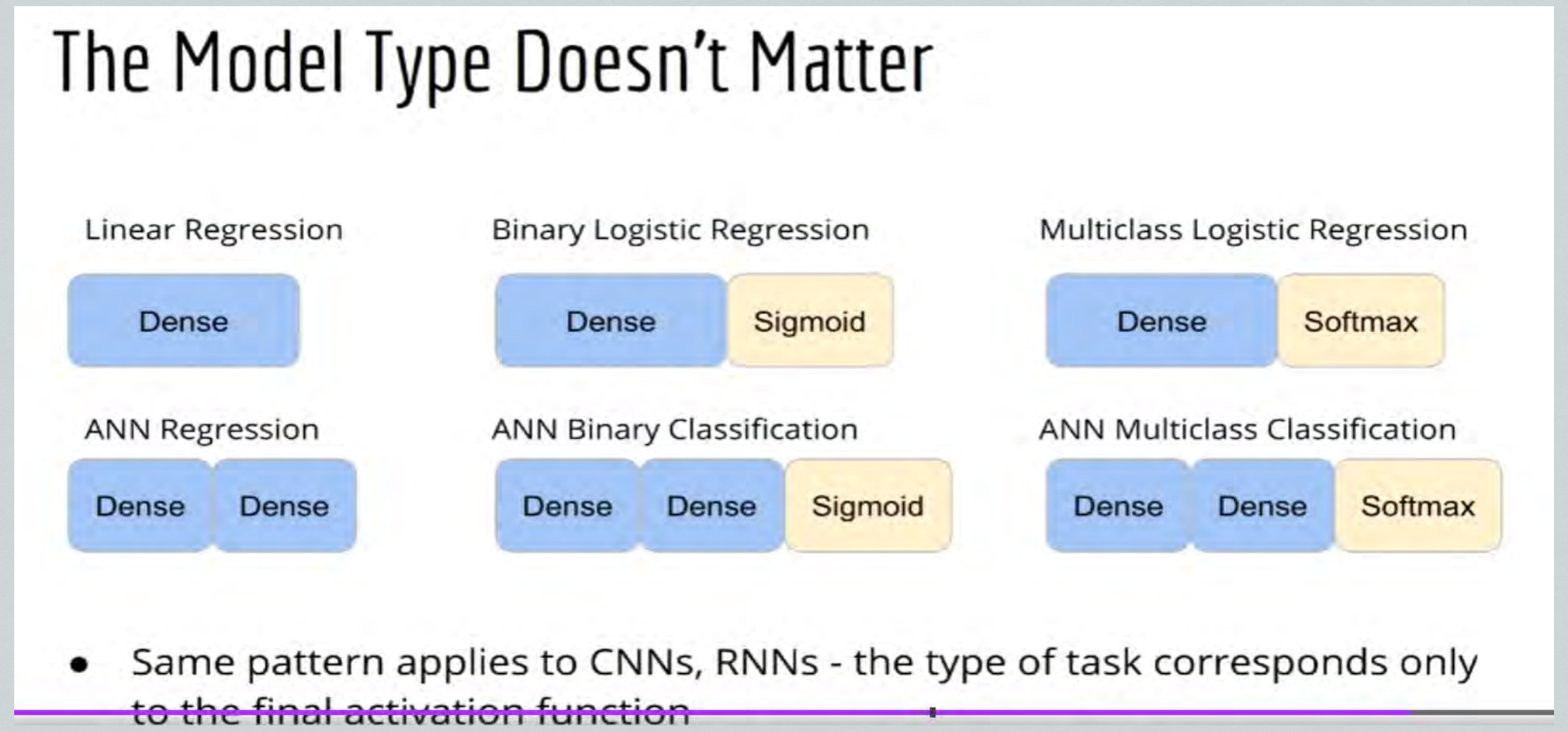
↳ This combines softmax + crossentropy

→ Multiclass Logistic Regression w/ softmax.

Summary So far

Task	activation function
Regression	none / identity
Binary class	Sigmoid
Multiclass	Softmax

→ function that Returns its input
→ Need 1 output node (Bernoulli dist)
→ k output Nodes



The Model Type Doesn't Matter

Linear Regression
Dense
Binary Logistic Regression
Dense
Sigmoid
Multiclass Logistic Regression
Dense
Softmax
ANN Regression
Dense
Dense
ANN Binary Classification
Dense
Dense
Sigmoid
ANN Multiclass Classification
Dense
Dense
Softmax

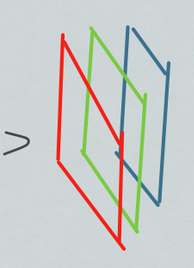
- Same pattern applies to CNNs, RNNs - the type of task corresponds only to the final activation function

Softmax can also handle k=2 (Binary but this is Redundant)

Lecture 28: How to Represent Images

Focus on the data: unstructured data.

Images (R,G,B) → A(i,j,k) →



I = width
J = height
k = color

Quantization
Range = 0 → 255
$2^8 2^8 2^8$ = 16.8 million colors

Grayscale
2D 0 = Black 255 = white

Images and neural networks
Neural networks don't like the large range of 0-255
→ We should scale them from 0-1
→ They are not centered around 0 this is an exception
↳ we don't standardize around 0 because this way the values can be interpreted as probabilities

VGG → Another exception

- Famous neural network in computer vision
- applications
  - ↳ Image classification
  - ↳ Transfer learning
  - ↳ Style transfer
  - ↳ object detection

→ Does not scale Image data But centers around 0
↳ Range still 256

How to input an Image?

Normaly an tensor is NxD (2D) but an Image is 3D array so convert to vector → Flattening
↳ view()
↳ Reshape()

Image (grayscale)

Pixel 1	Pixel 2
Pixel 3	pixel 4

vector

pixel 1	pixel 2	pixel 3	pixel 4

All data is the same Principle

Lecture 29: Code prep (ANN)

Problem: Handwriting recognition of Numbers → Multiclass classification

Recap of steps

1. Load in data
   - ↳ MNIST data
   - ↳ Grayscale
2. Build the Model
   - ↳ sequential Dense layers
   - ↳ End with multiclass logistic Regression
3. Train the Model
   - ↳ same + Batch gradient descent
4. Evaluate Model
   - ↳ same (Model agnostic)
5. Make predictions
   - ↳ see what is predicted wrong

Code

```
Train_dataset = torchvision.datasets.MNIST(
                root='.',
                train=true,
                download=true)
x_train = train_dataset.data
y_train = train_dataset.targets
```

→ N·28·28
→ N ↳ 3D ⚠

N_train = 60k
N_test = 10k
↳ train to False

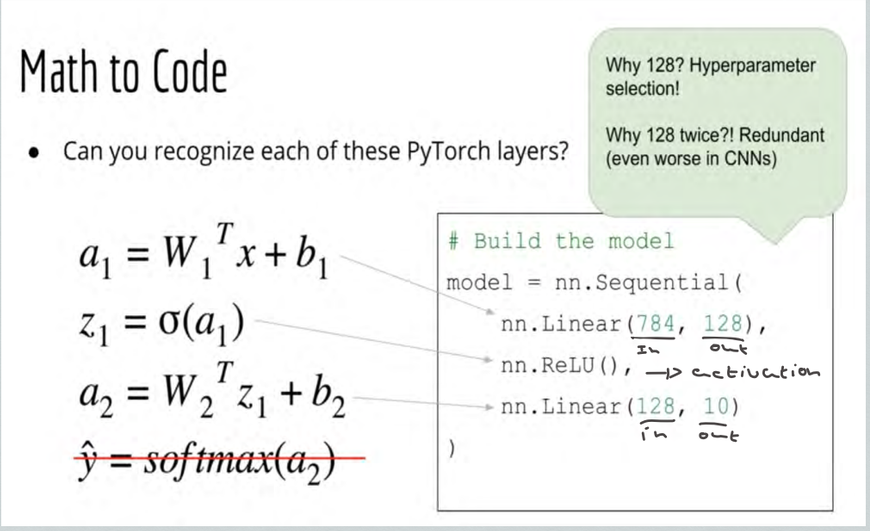
Scale the data and Make 2D

```
inputs = inputs.view(-1, 784)
```

σ is not always sigmoid

Build the Model

```
model = nn.Sequential(
        nn.Linear(784,128),
        nn.ReLU(),
        nn.Linear(128,10)
        )
```



Math to Code

- Can you recognize each of these PyTorch layers?

$$a_1 = W_1{}^T x + b_1$$
$$z_1 = \sigma(a_1)$$
$$a_2 = W_2{}^T z_1 + b_2$$

~~$\hat{y} = softmax(a_2)$~~

```
# Build the model
model = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

Why 128? Hyperparameter selection!

Why 128 twice?! Redundant (even worse in CNNs)

Double 128 is needed (output → Inp) and the have to be the same

Train the Model

Datasets are large (too much Ram)
Solution: Batch gradient descent
splits sets up in batches that do fit in Memory

Pseudocode:

```
For epoch in range(epochs):
    For x_batch, y_batch in batches(x, y, batch_size = 128):
        train(x_batch, y_batch)
        ↳ zeroing gradient,
          calculating loss
```

Data Loader Class.

```
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           Batch_size = batch_size,
                                           shuffle = True)
```

This Batching works because →
You take Random samples this works because they represent the population

Evaluate and Predict.

Because of batches it's harder to calc acc.

Code

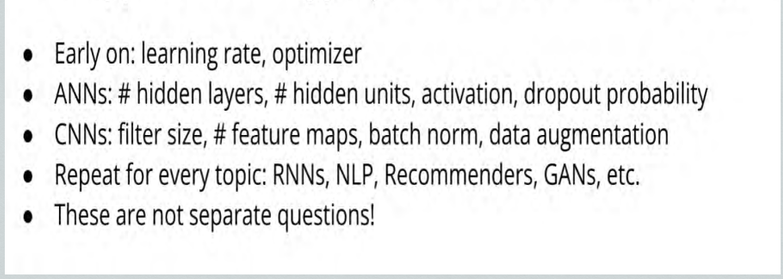
```
n_correct = 0
n_total = 0
for Inputs, targets in test_loader:
    # data to gpu
    Inputs, targets = inputs.to(device), targets.to(device)

    # Reshape the Input
    inputs = inputs.view(-1, 784)

    # Forward pass
    outputs = model(inputs)

    # get prediction
    _, predictions = torch.max(outputs, 1)

    # update counts
    n_correct = n_correct + (predictions == targets).sum().item()
    n_total += targets.shape[0]
test_acc = n_correct / n_total
```

Lecture 31: ANN For Regression

→ uses synthetic data so you know the answer

→ Can't play with the plot in notebook on local machine you can rotate + zoom

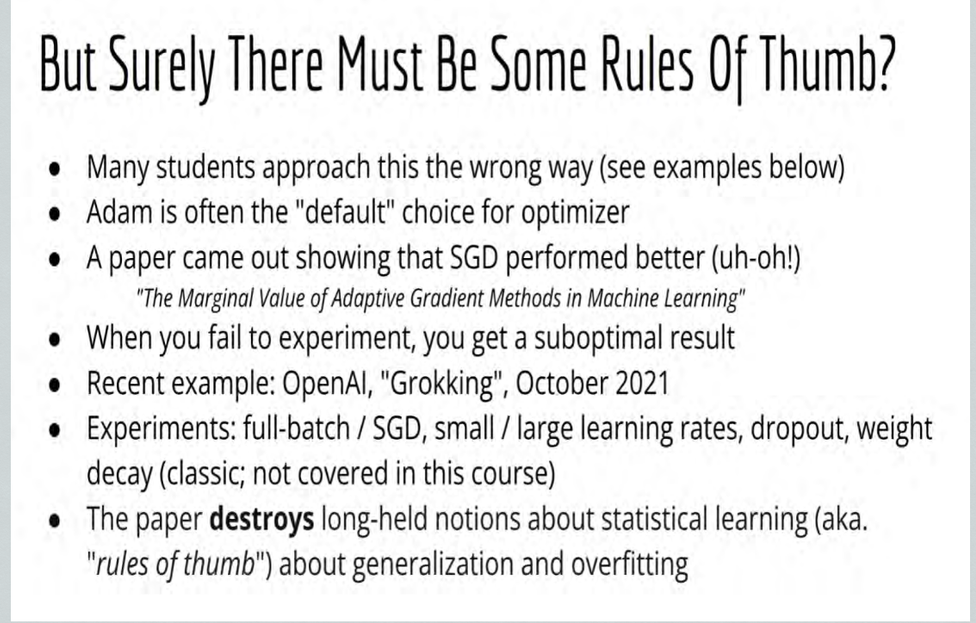
Lecture 32: How to choose Hyperparameters



This Applies To All Hyperparameters In This Course

- Early on: learning rate, optimizer
- ANNs: # hidden layers, # hidden units, activation, dropout probability
- CNNs: filter size, # feature maps, batch norm, data augmentation
- Repeat for every topic: RNNs, NLP, Recommenders, GANs, etc.
- These are not separate questions!

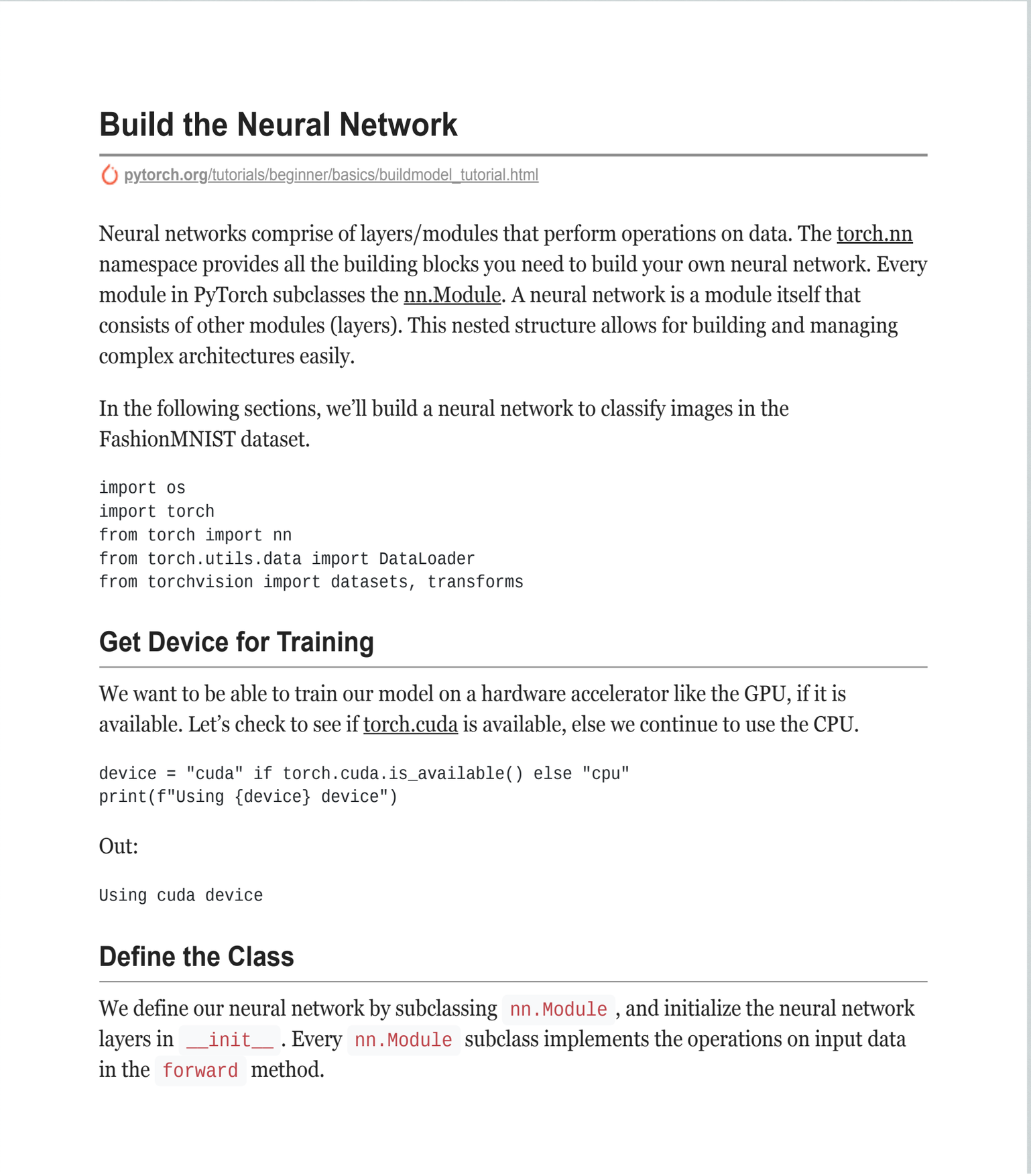
Machine learning is experimentation
→ Look up paper in the field
↳ can introduce bias
↳ Random search / grid search



But Surely There Must Be Some Rules Of Thumb?

- Many students approach this the wrong way (see examples below)
- Adam is often the "default" choice for optimizer
- A paper came out showing that SGD performed better (uh-oh!)
  "The Marginal Value of Adaptive Gradient Methods in Machine Learning"
- When you fail to experiment, you get a suboptimal result
- Recent example: OpenAI, "Grokking", October 2021
- Experiments: full-batch / SGD, small / large learning rates, dropout, weight decay (classic; not covered in this course)
- The paper **destroys** long-held notions about statistical learning (aka. "*rules of thumb*") about generalization and overfitting

https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html



Build the Neural Network

pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

Neural networks comprise of layers/modules that perform operations on data. The torch.nn namespace provides all the building blocks you need to build your own neural network. Every module in PyTorch subclasses the nn.Module. A neural network is a module itself that consists of other modules (layers). This nested structure allows for building and managing complex architectures easily.

In the following sections, we'll build a neural network to classify images in the FashionMNIST dataset.

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let's check to see if torch.cuda is available, else we continue to use the CPU.

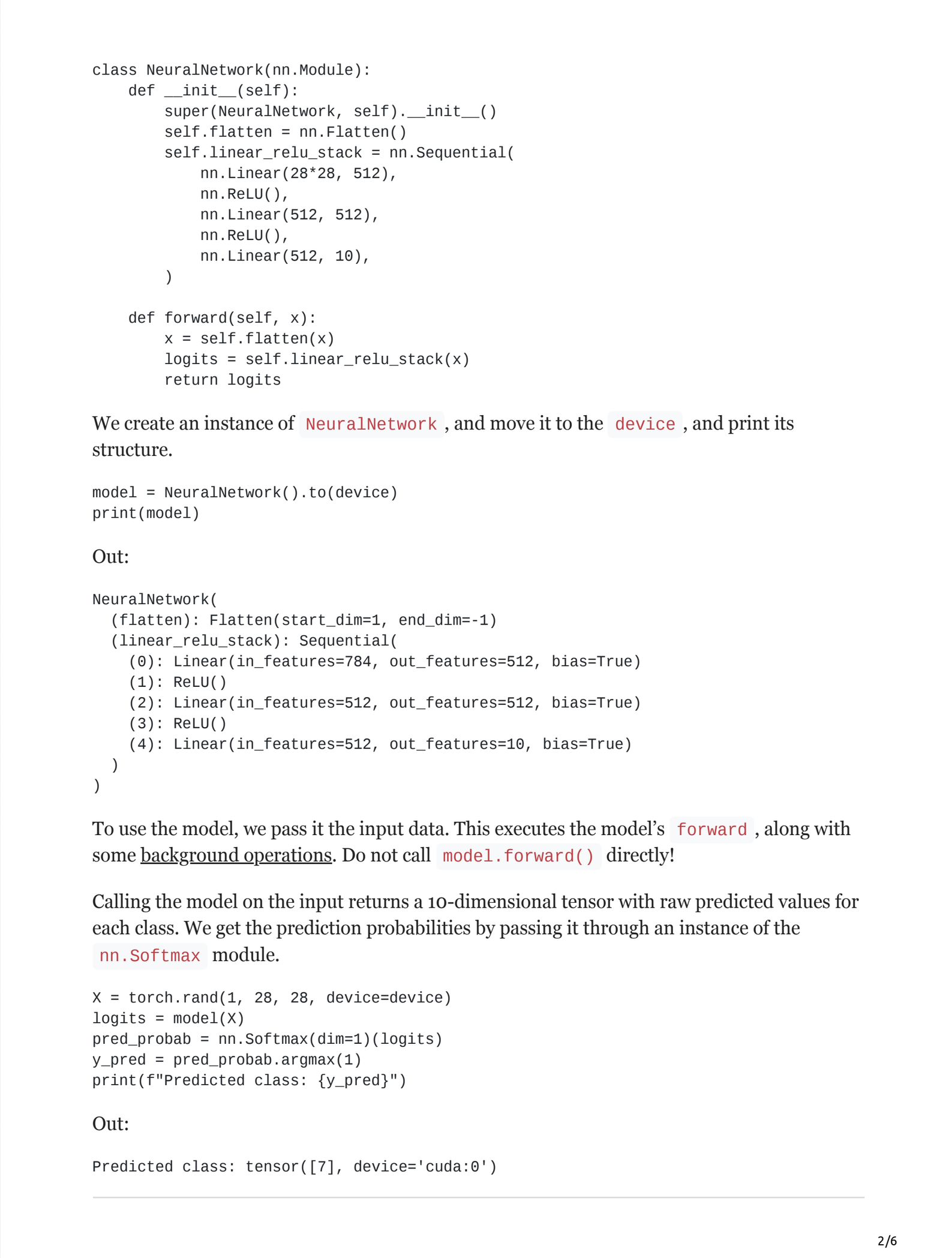
```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

Out:

```
Using cuda device
```

Define the Class

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.



```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

We create an instance of `NeuralNetwork`, and move it to the `device`, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

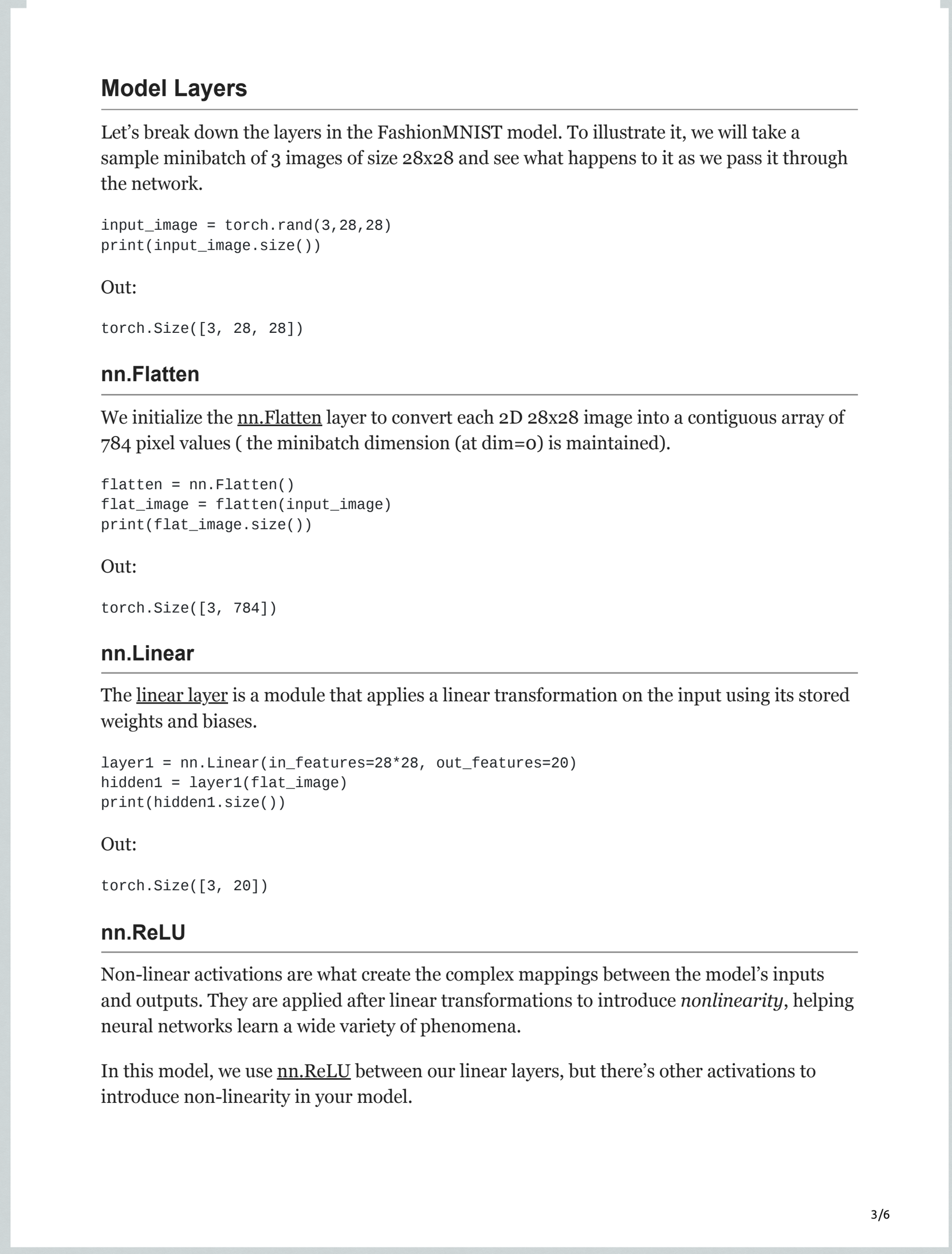
To use the model, we pass it the input data. This executes the model's `forward`, along with some background operations. Do not call `model.forward()` directly!

Calling the model on the input returns a 10-dimensional tensor with raw predicted values for each class. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([7], device='cuda:0')
```



Model Layers

Let's break down the layers in the FashionMNIST model. To illustrate it, we will take a sample minibatch of 3 images of size 28x28 and see what happens to it as we pass it through the network.

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

Out:

```
torch.Size([3, 28, 28])
```

nn.Flatten

We initialize the nn.Flatten layer to convert each 2D 28x28 image into a contiguous array of 784 pixel values (the minibatch dimension (at dim=0) is maintained).

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

Out:

```
torch.Size([3, 784])
```

nn.Linear

The linear layer is a module that applies a linear transformation on the input using its stored weights and biases.

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

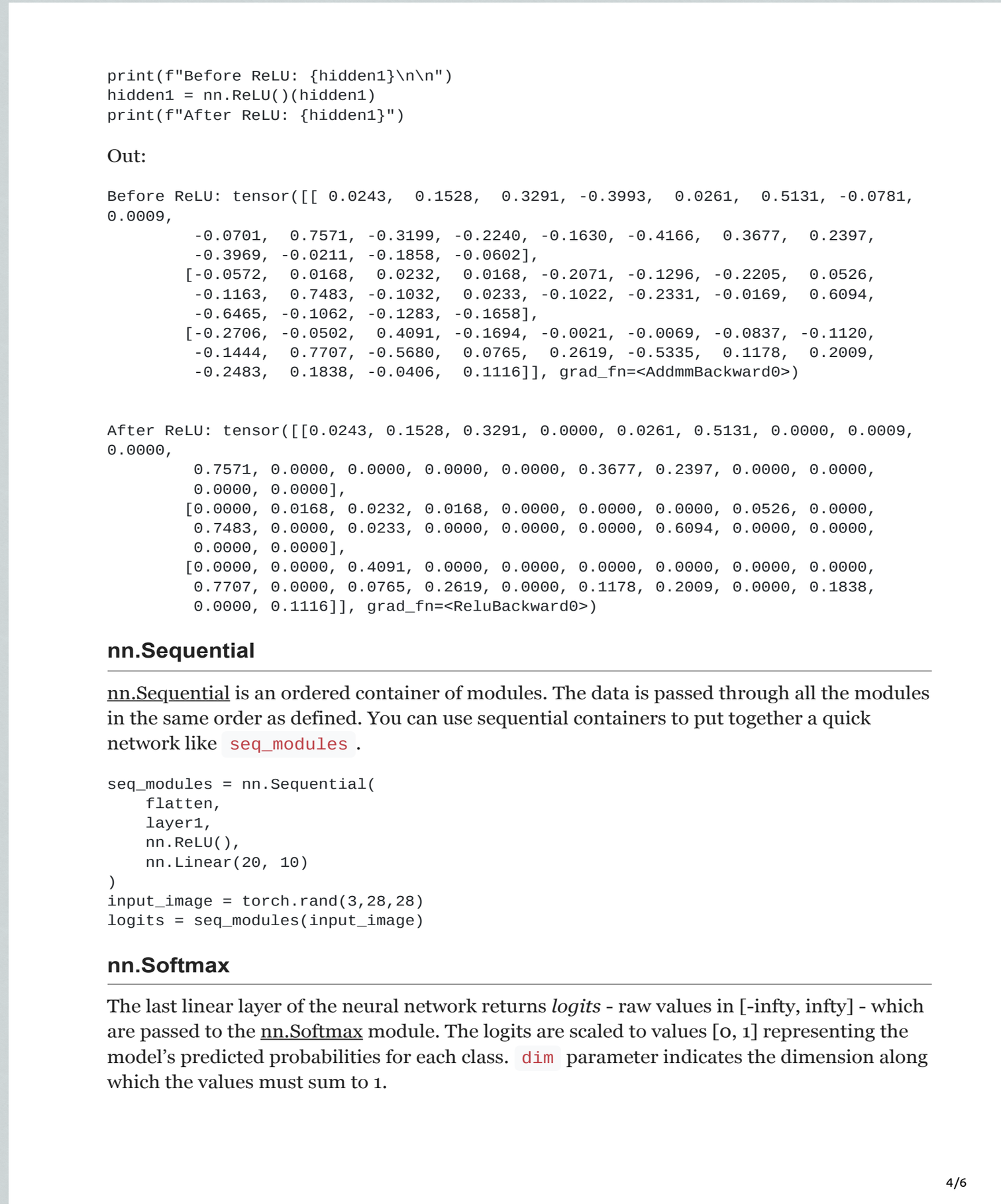
Out:

```
torch.Size([3, 20])
```

nn.ReLU

Non-linear activations are what create the complex mappings between the model's inputs and outputs. They are applied after linear transformations to introduce *nonlinearity*, helping neural networks learn a wide variety of phenomena.

In this model, we use nn.ReLU between our linear layers, but there's other activations to introduce non-linearity in your model.



```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

Out:

```
Before ReLU: tensor([[ 0.0243,  0.1528,  0.3291, -0.3993,  0.0261,  0.5131, -0.0781,
0.0009,
         -0.0701,  0.7571, -0.3199, -0.2240, -0.1630, -0.4166,  0.3677,  0.2397,
         -0.3969, -0.0211, -0.1858, -0.0602],
        [-0.0572,  0.0168,  0.0232,  0.0168, -0.2071, -0.1296, -0.2205,  0.0526,
         -0.1163,  0.7483, -0.1032,  0.0233, -0.1022, -0.2331, -0.0169,  0.6094,
         -0.6465, -0.1062, -0.1283, -0.1658],
        [-0.2706, -0.0502,  0.4091, -0.1694, -0.0021, -0.0069, -0.0837, -0.1120,
         -0.1444,  0.7707, -0.5680,  0.0765,  0.2619, -0.5335,  0.1178,  0.2009,
         -0.2483,  0.1838, -0.0406,  0.1116]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.0243, 0.1528, 0.3291, 0.0000, 0.0261, 0.5131, 0.0000, 0.0009,
0.0000,
         0.7571, 0.0000, 0.0000, 0.0000, 0.0000, 0.3677, 0.2397, 0.0000, 0.0000,
         0.0000, 0.0000],
        [0.0000, 0.0168, 0.0232, 0.0168, 0.0000, 0.0000, 0.0000, 0.0526, 0.0000,
         0.7483, 0.0000, 0.0233, 0.0000, 0.0000, 0.0000, 0.6094, 0.0000, 0.0000,
         0.0000, 0.0000],
        [0.0000, 0.0000, 0.4091, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
         0.7707, 0.0000, 0.0765, 0.2619, 0.0000, 0.1178, 0.2009, 0.0000, 0.1838,
         0.0000, 0.1116]], grad_fn=<ReluBackward0>)
```

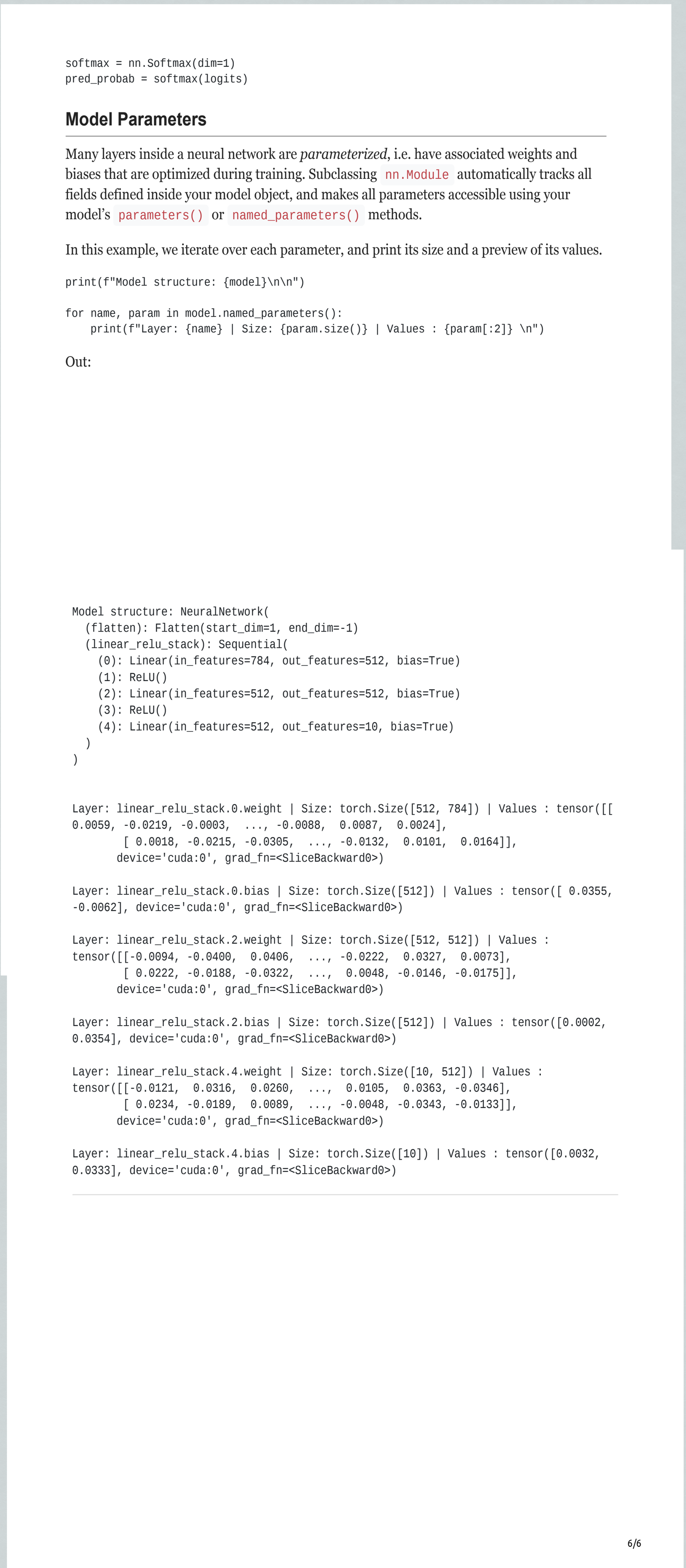
nn.Sequential

nn.Sequential is an ordered container of modules. The data is passed through all the modules in the same order as defined. You can use sequential containers to put together a quick network like `seq_modules`.

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

nn.Softmax

The last linear layer of the neural network returns *logits* - raw values in [-infty, infty] - which are passed to the nn.Softmax module. The logits are scaled to values [0, 1] representing the model's predicted probabilities for each class. `dim` parameter indicates the dimension along which the values must sum to 1.



```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

Model Parameters

Many layers inside a neural network are *parameterized*, i.e. have associated weights and biases that are optimized during training. Subclassing `nn.Module` automatically tracks all fields defined inside your model object, and makes all parameters accessible using your model's `parameters()` or `named_parameters()` methods.

In this example, we iterate over each parameter, and print its size and a preview of its values.

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

Out:

```
Model structure: NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)


Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[
0.0059, -0.0219, -0.0003,  ..., -0.0088,  0.0087,  0.0024],
        [ 0.0018, -0.0215, -0.0305,  ..., -0.0132,  0.0101,  0.0164]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0355,
-0.0062], device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values :
tensor([[-0.0094, -0.0400,  0.0406,  ..., -0.0222,  0.0327,  0.0073],
        [ 0.0222, -0.0188, -0.0322,  ...,  0.0048, -0.0146, -0.0175]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([0.0002,
0.0354], device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values :
tensor([[-0.0121,  0.0316,  0.0260,  ...,  0.0105,  0.0363, -0.0346],
        [ 0.0234, -0.0189,  0.0089,  ..., -0.0048, -0.0343, -0.0133]],
       device='cuda:0', grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([0.0032,
0.0333], device='cuda:0', grad_fn=<SliceBackward0>)
```