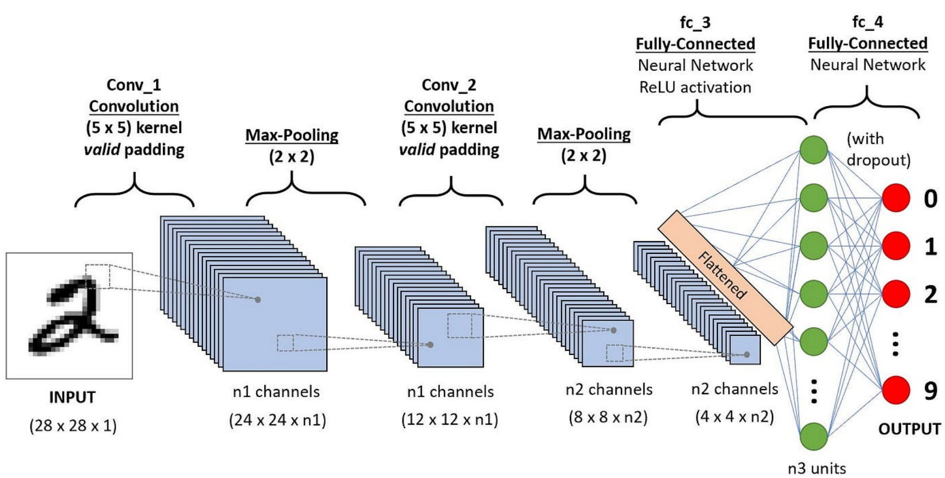
```
@bind (macro with 1 method)
```

```julia
begin
    ### A Pluto.jl notebook ###
    # v0.19.29

    using Markdown
    using InteractiveUtils

    # This Pluto notebook uses @bind for interactivity. When running this notebook outside of Pluto, the following 'mock version' of @bind gives bound variables a default value (instead of an error).
    macro bind(def, element)
        quote
            local iv = try Base.loaded_modules[Base.PkgId(Base.UUID("6e696c72-6542-2067-7265-42206c756150"), "AbstractPlutoDingetjes")].Bonds.initial_value catch; b -> missing; end
            local el = $(esc(element))
            global $(esc(def)) = Core.applicable(Base.get, el) ? Base.get(el) : iv(el)
            el
        end
    end
end
```

Table of Contents


```julia
begin
    using PlutoUI
    using Latexify
    TableOfContents()
end
```

```julia
using Flux: crossentropy, Momentum
```




```julia
md"""
![CNN_1](https://user-images.githubusercontent.com/1801654/224518240-bfe1ea42-3ba4-474c-9e11-e68c1da760be.jpg)

"""
```

Training a CNN classifier using `Flux`

MNIST is a dataset of 60k small training images of handwritten digits from 0 to 9.

We will do the following steps in order:

- Load MNIST training and test datasets
- Define a Convolution Neural Network (CNN)
- Define a loss function
- Train the network on the training data
- Test the network on the test data

```
md"""
## Training a CNN classifier using `Flux`

[MNIST](https://www.cs.toronto.edu/~kriz/mnist.html) is a dataset of 60k small training images of handwritten digits from 0 to 9.

We will do the following steps in order:

- Load MNIST training and test datasets
- Define a Convolution Neural Network (CNN)
- Define a loss function
- Train the network on the training data
- Test the network on the test data

"""
```

```

```

```
begin
    using Statistics
    using CUDA
    using Flux, Flux.Optimise
    using MLDatasets: CIFAR10
    using Images.ImageCore
    using Flux: onehotbatch, onecold
    using Base.Iterators: partition

    using Plots
end
```

```
Package cuDNN not found in current path.
- Run `import Pkg; Pkg.add("cuDNN")` to install the cuDNN package, then restart julia.
- If cuDNN is not installed, some Flux functionalities will not be available when running on the GPU.
```

> **Tip**
>
> The preceding cell needs to be modified to read in the CIFAR10 database rather than MNIST.

```
md"""
!!! tip
    The preceding cell needs to be modified to read in the CIFAR10 database rather than MNIST.
"""
```

Set an environment variable to stop the system asking for approval to download data.

```
md"""
Set an environment variable to stop the system asking for approval to download data.
"""
```

```
ENV["DATADEPS_ALWAYS_ACCEPT"] = true;
```

```
10×50000 OneHotMatrix(::Vector{UInt32}) with eltype Bool:
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  …  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  1  1
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1     1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  …  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1  1  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅
```

```julia
begin
    train_x, train_y = CIFAR10(split=:train)[:]
    train_x = reshape(train_x, 32, 32, 3, :)
    labels2 = onehotbatch(train_y, 0:9)
end
```

```
7-Zip 21.07 (x64) : Copyright (c) 1999-2021 Igor Pavlov : 2021-12-26


Extracting archive:
--
Path =
Type = tar
Code Page = UTF-8
Characteristics = ASCII

Everything is Ok

Folders: 1
Files: 8
Size:       184380149
Compressed: 9216
```

The train_x array contains 60,000 28x28 pixel images converted to 28 x 28 x 1 arrays with the third dimension acting as a grey-scale channel. Let's take a look at a random image from train_x. For this, we need to convert the floats in the 28x28 matrix to the grey colour value:

```julia
md"""
The train\_x array contains 60,000 28x28 pixel images converted to 28 x 28 x 1 arrays with the third dimension acting as a grey-scale channel. Let's take a look at a random image from train_x. For this, we need to convert the floats in the 28x28 matrix to the grey colour value:
"""
```

Tip

The size and dimensionality of the CIFAR10 data base are different: there are 50,000 images of 32 x 32 pixels involving 3 colour channels

```julia
md"""
!!! tip
    The size and dimensionality of the CIFAR10 data base are different: there are 50,000 images of 32 x 32 pixels involving 3 colour channels
"""
```

```
3×32×32 Array{Gray{Float32},3} with eltype Gray{Float32}:
[:, :, 1] =
 Gray{Float32}(0.917647)  Gray{Float32}(0.556863)  …  Gray{Float32}(0.619608)
 Gray{Float32}(0.92549)   Gray{Float32}(0.552941)     Gray{Float32}(0.513726)
 Gray{Float32}(0.901961)  Gray{Float32}(0.521569)     Gray{Float32}(0.411765)

[:, :, 2] =
 Gray{Float32}(0.839216)  Gray{Float32}(0.435294)  …  Gray{Float32}(0.607843)
 Gray{Float32}(0.843137)  Gray{Float32}(0.435294)     Gray{Float32}(0.498039)
 Gray{Float32}(0.831373)  Gray{Float32}(0.415686)     Gray{Float32}(0.439216)

[:, :, 3] =
 Gray{Float32}(0.717647)  Gray{Float32}(0.45098)   …  Gray{Float32}(0.533333)
 Gray{Float32}(0.713726)  Gray{Float32}(0.45098)      Gray{Float32}(0.411765)
 Gray{Float32}(0.717647)  Gray{Float32}(0.443137)     Gray{Float32}(0.345098)

;;; …

[:, :, 30] =
 Gray{Float32}(0.372549)  Gray{Float32}(0.368627)  …  Gray{Float32}(0.772549)
 Gray{Float32}(0.309804)  Gray{Float32}(0.294118)     Gray{Float32}(0.788235)
 Gray{Float32}(0.160784)  Gray{Float32}(0.152941)     Gray{Float32}(0.690196)

[:, :, 31] =
 Gray{Float32}(0.490196)  Gray{Float32}(0.501961)  …  Gray{Float32}(0.701961)
 Gray{Float32}(0.407843)  Gray{Float32}(0.396078)     Gray{Float32}(0.701961)
 Gray{Float32}(0.294118)  Gray{Float32}(0.286275)     Gray{Float32}(0.611765)

[:, :, 32] =
 Gray{Float32}(0.780392)  Gray{Float32}(0.627451)  …  Gray{Float32}(0.545098)
 Gray{Float32}(0.709804)  Gray{Float32}(0.52549)      Gray{Float32}(0.556863)
 Gray{Float32}(0.635294)  Gray{Float32}(0.45098)      Gray{Float32}(0.462745)
```

```julia
Gray.(permutedims(reshape(train_x[:,:,:,rand(1:end)], 32, 32, 3), (3, 2, 1)))

```

Tip

Use this function: `image(x) = colorview(RGB, permutedims(x, (3, 2, 1)))` to display the colour images instead of `Gray`.

```julia
md"""
!!! tip
    Use this function: `image(x) = colorview(RGB, permutedims(x, (3, 2, 1)))` to display the colour images instead of `Gray`.
"""
```

The images are simply 28 x 28 matrices of numbers plus one greyscale channel. We can now arrange them in batches of say, 1,000 and keep a validation set to track our progress. This process is called minibatch learning, which is a popular method of training large neural networks. Rather than sending the entire dataset at once, we break it down into smaller chunks (called minibatches) typically chosen at random.

The first 59k images (in batches of 1,000) will be our training set, and the rest are for validation used to track training progress. `partition` handily breaks down the set we give it in consecutive parts (1,000 in this case).

```julia
md"""
The images are simply 28 x 28 matrices of numbers plus one greyscale channel. We can now arrange them in batches of say, 1,000 and keep a validation set to track our progress.  This process is called minibatch learning, which is a popular method of training large neural networks. Rather than sending the entire dataset at once, we break it down into smaller chunks (called minibatches) typically chosen at random.

The first 59k images (in batches of 1,000) will be our training set, and the rest are for validation used to track training progress. `partition` handily breaks down the set we give it in consecutive parts (1,000 in this case).
"""
```

```julia
begin

    # Function to convert images for display
    image(x) = colorview(RGB, permutedims(x, (3, 2, 1)))

    # Organize into minibatches (each containing 1000 images)
    batch_size = 1000
    num_batches = div(size(train_x, 4), batch_size)

    # Split into training and validation sets
    train_set = [train_x[:, :, :, (i-1)*batch_size+1:i*batch_size] for i in 1:num_batches-1]
    validation_set = train_x[:, :, :, end-(batch_size-1):end]

    # Display the first image in the validation set as an example
    image(validation_set[:, :, :, 20])

end
```

Tip

Because CIFAR10 comprises 50,000 images rather than 60,000, the split between training and validation needs to reflect this.

```julia
md"""
!!! tip
    Because CIFAR10 comprises 50,000 images rather than 60,000, the split between training and validation needs to reflect this.
"""
```

```
10×5000 OneHotMatrix(::Vector{UInt32}) with eltype Bool:
 ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  …  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1     ⋅  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  1  1
 ⋅  ⋅  ⋅  ⋅  ⋅  1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  …  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅
 1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅     ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  ⋅
```

```julia
begin
    train = ([(train_x[:,:,:,i], labels2[:,i]) for i in partition(1:50000, 1000)]) |> gpu
    valset = 45001:50000
    valX = train_x[:,:,:,valset] |> gpu
    valY = labels2[:, valset] |> gpu

end
```

```
The CUDA function is being called but CUDA.jl is not functional.
Defaulting back to the CPU. (No action is required if you want to run on the CP
U).
```

Defining the Classifier

Now we can define our Convolutional Neural Network (CNN).

A convolutional neural network is one which defines a kernel and slides it across a matrix to create an intermediate representation to extract features from. It creates higher order features as it goes into deeper layers, making it suitable for images, where the structure of the subject is what will help us determine which class it belongs to.

```
md"""
### Defining the Classifier

Now we can define our Convolutional Neural Network (CNN).

A convolutional neural network is one which defines a kernel and slides it across a matrix to create an intermediate representation to extract features from. It creates higher order features as it goes into deeper layers, making it suitable for images, where the structure of the subject is what will help us determine which class it belongs to.
"""
```

```
m3 = Chain(
       Conv((5, 5), 3 => 32, relu, pad=2),   # 2_432 parameters
       MaxPool((2, 2)),
       Conv((5, 5), 32 => 16, relu, pad=2),  # 12_816 parameters
       MaxPool((2, 2)),
       Main.var"#11#12"{typeof(reshape), typeof(size), Colon}(reshape, size, Colon()),
       Dense(1024 => 120),                   # 123_000 parameters
       Dense(120 => 84),                     # 10_164 parameters
       Dense(84 => 10),                      # 850 parameters
       NNlib.softmax,
     )                   # Total: 10 arrays, 149_262 parameters, 584.289 KiB.
```

```
# Define the neural network architecture
m3 = Chain(
    Conv((5, 5), 3 => 32, pad=(2, 2), relu),  # Modify input filter number from 1 to 3 for 3 input channels
    MaxPool((2, 2)),
    Conv((5, 5), 32 => 16, pad=(2, 2), relu), # Modify output channels from 16 to 32
    MaxPool((2, 2)),
    x -> reshape(x, :, size(x, 4)),
    Dense(1024, 120),  # Modify input dimension from 392 to 1024 for the first Dense layer
    Dense(120, 84),
    Dense(84, 10),
    softmax
) |> gpu
```

Tip

You need to modify the input filter number of the first `Conv` layer and the input dimension of the first `Dense` layer.

```
md"""
!!! tip
    You need to modify the input filter number of the first `Conv` layer and the input dimension of the first `Dense` layer.
"""
```

We will use a crossentropy loss and the Momentum optimiser here. Crossentropy will be a good option when it comes to working with multiple independent classes. Momentum smooths out the noisy gradients and helps towards a smooth convergence. Gradually lowering the learning rate along with momentum helps to maintain a bit of adaptivity in our optimisation, preventing us from overshooting our desired destination.

```
md"""
We will use a crossentropy loss and the Momentum optimiser here. Crossentropy will be a good option when it comes to working with multiple independent classes. Momentum smooths out the noisy gradients and helps towards a smooth convergence. Gradually lowering the learning rate along with momentum helps to maintain a bit of adaptivity in our optimisation, preventing us from overshooting our desired destination.
"""
```

```
Momentum(0.01, 0.9, IdDict())
```

```julia
begin
    loss2(x, y) = sum(crossentropy(m3(x), y))
    opt2 = Momentum(0.01)
end
```

We can start writing our train loop where we will keep track of some basic accuracy numbers about our model. We can define an `accuracy` function for it like so:

```julia
md"""
We can start writing our train loop where we will keep track of some basic accuracy numbers about our model. We can define an `accuracy` function for it like so:
"""
```

```
accuracy (generic function with 1 method)
```

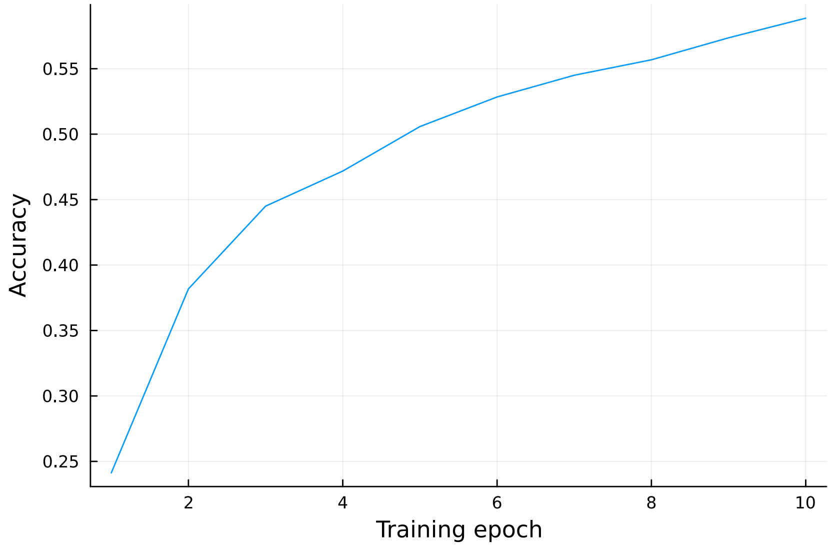
```julia
accuracy(x, y) = mean(onecold(m3(x), 0:9) .== onecold(y, 0:9))
```

Training the network

Training is where we do a bunch of the interesting operations we defined earlier, and see what our net is capable of. We will loop over the dataset 10 times and feed the inputs to the neural network and optimise.

```julia
md"""
### Training the network

Training is where we do a bunch of the interesting operations we defined earlier, and see what our net is capable of. We will loop over the dataset 10 times and feed the inputs to the neural network and optimise.
"""
```

```julia
begin
    train_acc = Float32[]
    train_epochs = Int32[]
    epochs = 10

    for epoch = 1:epochs
        for d in train
            gs = gradient(Flux.params(m3)) do
                l = loss2(d...)
            end
            update!(opt2, Flux.params(m3), gs)
        end
        push!(train_acc, accuracy(valX, valY))
        push!(train_epochs, epoch)
    end

end
```




```
begin
    plot(train_epochs, train_acc, lab="")
    yaxis!("Accuracy")
    xaxis!("Training epoch")
end
```

> **Tip**
>
> The training regime doesn't need modification.

```
md"""
!!! tip
    The training regime doesn't need modification.
"""
```

Testing the network

We have trained the network for 10 passes over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions. This will be done on a yet unseen section of data.

First step. Let us perform the exact same preprocessing on this set, as we did on our training set.

```
md"""
### Testing the network

We have trained the network for 10 passes over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions. This will be done on a yet unseen section of data.

First step. Let us perform the exact same preprocessing on this set, as we did on our training set.

"""
```

Next, display images from the test set.

```
md"""
Next, display images from the test set.
"""
```

```
32×32×3×10000 Array{Float32, 4}:
[:, :, 1, 1] =
 0.619608  0.596078  0.592157  0.607843  0.607843  …  0.266667   0.239216   0.211765
 0.623529  0.592157  0.592157  0.607843  0.611765     0.164706   0.192157   0.219608
 0.647059  0.623529  0.619608  0.627451  0.631373     0.121569   0.137255   0.176471
 0.65098   0.65098   0.654902  0.682353  0.666667     0.14902    0.168627   0.168627
 0.627451  0.635294  0.627451  0.654902  0.662745     0.145098   0.152941   0.156863
 0.611765  0.627451  0.639216  0.654902  0.639216  …  0.168627   0.164706   0.156863
 0.635294  0.643137  0.647059  0.662745  0.662745     0.164706   0.172549   0.156863
 ⋮                                                 ⋱             ⋮
 0.54902   0.568627  0.560784  0.443137  0.333333     0.309804   0.427451   0.411765
 0.552941  0.556863  0.54902   0.517647  0.411765     0.286275   0.364706   0.34902
 0.560784  0.560784  0.556863  0.54902   0.501961     0.219608   0.235294   0.188235
 0.537255  0.533333  0.545098  0.54902   0.541176     0.14902    0.101961   0.0941176
 0.494118  0.490196  0.509804  0.533333  0.521569  …  0.0509804  0.113725   0.133333
 0.454902  0.466667  0.470588  0.498039  0.505882     0.156863   0.0784314  0.0823529

[:, :, 2, 1] =
 0.439216  0.439216  0.431373  0.419608  …  0.423529  0.486275  0.454902  0.419608
 0.435294  0.431373  0.427451  0.431373     0.380392  0.392157  0.4       0.411765
 0.454902  0.447059  0.435294  0.427451     0.360784  0.345098  0.333333  0.34902
 0.462745  0.454902  0.435294  0.439216     0.345098  0.356863  0.356863  0.337255
 0.439216  0.439216  0.415686  0.431373     0.345098  0.341176  0.352941  0.34902
 0.427451  0.443137  0.45098   0.458824  …  0.329412  0.34902   0.360784  0.360784
 0.45098   0.458824  0.458824  0.470588     0.521569  0.309804  0.345098  0.341176
 ⋮                                       ⋱                      ⋮
 0.384314  0.4       0.403922  0.333333     0.4       0.517647  0.611765  0.572549
 0.380392  0.380392  0.388235  0.384314     0.490196  0.513726  0.568627  0.529412
 0.380392  0.384314  0.388235  0.4          0.431373  0.454902  0.45098   0.388235
 0.372549  0.372549  0.384314  0.396078     0.352941  0.380392  0.321569  0.301961
 0.356863  0.356863  0.372549  0.388235  …  0.235294  0.25098   0.321569  0.329412
 0.333333  0.345098  0.34902   0.368627     0.364706  0.333333  0.25098   0.262745
```

```julia
begin
    test_x, test_y = CIFAR10(split=:test)[:]
    test_x = reshape(test_x, 32, 32, 3, :)
end
```

```julia
test_labels = onehotbatch(test_y, 0:9);
```

```
test =
  [(32×32×3×1000 Array{Float32, 4}:
    [:, :, 1, 1] =
```

```julia
test = ([(test_x[:,:,:,i], test_labels[:,i]) for i in partition(1:10000, 1000)]) |>
gpu
```

```julia
@bind image_index Slider(1:100:10000, default=1)
```

```
3×32×32 Array{Gray{Float32},3} with eltype Gray{Float32}:
[:, :, 1] =
 Gray{Float32}(0.619608)  Gray{Float32}(0.596078)  …  Gray{Float32}(0.211765)
 Gray{Float32}(0.439216)  Gray{Float32}(0.439216)     Gray{Float32}(0.419608)
 Gray{Float32}(0.192157)  Gray{Float32}(0.2)          Gray{Float32}(0.627451)

[:, :, 2] =
 Gray{Float32}(0.623529)  Gray{Float32}(0.592157)  …  Gray{Float32}(0.219608)
 Gray{Float32}(0.435294)  Gray{Float32}(0.431373)     Gray{Float32}(0.411765)
 Gray{Float32}(0.184314)  Gray{Float32}(0.156863)     Gray{Float32}(0.584314)

[:, :, 3] =
 Gray{Float32}(0.647059)  Gray{Float32}(0.623529)  …  Gray{Float32}(0.176471)
 Gray{Float32}(0.454902)  Gray{Float32}(0.447059)     Gray{Float32}(0.34902)
 Gray{Float32}(0.2)       Gray{Float32}(0.176471)     Gray{Float32}(0.517647)

;;; …

[:, :, 30] =
 Gray{Float32}(0.537255)  Gray{Float32}(0.533333)  …  Gray{Float32}(0.0941176)
 Gray{Float32}(0.372549)  Gray{Float32}(0.372549)     Gray{Float32}(0.301961)
 Gray{Float32}(0.141176)  Gray{Float32}(0.121569)     Gray{Float32}(0.486275)

[:, :, 31] =
 Gray{Float32}(0.494118)  Gray{Float32}(0.490196)  …  Gray{Float32}(0.133333)
 Gray{Float32}(0.356863)  Gray{Float32}(0.356863)     Gray{Float32}(0.329412)
 Gray{Float32}(0.141176)  Gray{Float32}(0.12549)      Gray{Float32}(0.505882)

[:, :, 32] =
 Gray{Float32}(0.454902)  Gray{Float32}(0.466667)  …  Gray{Float32}(0.0823529)
 Gray{Float32}(0.333333)  Gray{Float32}(0.345098)     Gray{Float32}(0.262745)
 Gray{Float32}(0.129412)  Gray{Float32}(0.133333)     Gray{Float32}(0.431373)
```

```julia
Gray.(permutedims(reshape(test_x[:,:,:,image_index], 32, 32,3), (3,2,1)))
```

The outputs are energies for the 10 classes. Higher the energy for a class, the more the network thinks that the image is of the particular class. Every column corresponds to the output of one image, with the 10 floats in the column being the energies.

Let's see how the model fared:

```
md"""
The outputs are energies for the 10 classes. Higher the energy for a class, the more the network thinks that the image is of the particular class. Every column corresponds to the output of one image, with the 10 floats in the column being the energies.

Let's see how the model fared:
"""
```

```
10×5 Matrix{Float32}:
 0.12386      0.160771    0.00802864  0.0013145    1.63209f-5
 6.27258f-5   0.0413777   0.00262401  0.000331806  3.84491f-6
 0.693682     0.00954001  0.0455476   0.0269539    0.0096018
 0.021898     0.0096466   0.167972    0.233842     0.144129
 0.0388883    0.00762291  0.119493    0.0296499    0.00107298
 0.0200257    0.00188005  0.127114    0.565769     0.834886
 0.000224681  0.00941081  0.428894    0.0660704    0.00732907
 0.083864     0.00113594  0.0326832   0.0698819    0.00280342
 0.0085742    0.66651     0.0641139   0.00232622   1.54209f-5
 0.00891988   0.0921051   0.00352822  0.00386034   0.00014131
```

```
begin
    ids = rand(1:10000, 5)
    rand_test  = test_x[:,:,:,ids] |> gpu
    rand_truth = test_y[ids]
    m3(rand_test)
end
```

Tip

For visualisation you should display performance on a random sample of test images, say 20, and set up a slider to navigate them while displaying the image, the ground truth label, and the label predicted by the network.

```
md"""
!!! tip
    For visualisation you should display performance on a random sample of test images, say 20, and set up a slider to navigate them while displaying the image, the ground truth label, and the label predicted by the network.
"""
```

This looks similar to how we would expect the results to be. At this point, it's a good idea to see how our net actually performs on new data, that we have prepared.

```
md"""
This looks similar to how we would expect the results to be. At this point, it's a good idea to see how our net actually performs on new data, that we have prepared.
"""
```

```
0.547
```

```
accuracy(test[1]...)
```

This is much better than random chance set at 10% (since we only have 10 classes), and not bad at all for a relatively small hand-coded network like this one.

```
md"""
This is much better than random chance set at 10% (since we only have 10 classes), and not bad at all for a relatively small hand-coded network like this one.
"""
```

Let's take a look at how the net performed on all the classes performed individually.

```
md"""
 Let's take a look at how the net performed on all the classes performed individually.
"""
```

```
begin
    class_correct = zeros(10)
    class_total = zeros(10)
    for i in 1:10
        preds = m3(test[i][1])
        lab = test[i][2]
        for j = 1:1000
            pred_class = findmax(preds[:, j])[2]
            actual_class = findmax(lab[:, j])[2]
            if pred_class == actual_class
                class_correct[pred_class] += 1
            end
            class_total[actual_class] += 1
        end
    end
end
```

	accuracy	label
1	0.532	0
2	0.648	1
3	0.37	2
4	0.39	3
5	0.373	4
6	0.52	5
7	0.752	6
8	0.656	7
9	0.728	8
10	0.653	9

```
begin
    using DataFrames
    DataFrame(accuracy=(class_correct ./ class_total), label=0:9)
end
```

Tip

For legibility you should assign labels to the image categories.

```
md"""
!!! tip
    For legibility you should assign labels to the image categories.
"""
```

The spread seems pretty good, with certain classes performing significantly better than the others.

```
md"""
The spread seems pretty good, with certain classes performing significantly better than the others.

"""
```

ENTS =
3ef-5323-5732-b1bb-66c8b64840ba\"\nDataFrames = \"a93c6f00-e57d-5684-b7b6-d8193f3e46c0\"\nFl

PLUTO_MANIFEST_TOML_CONTENTS =
"# This file is machine-generated - editing it directly is not advised\n\njulia_version = '