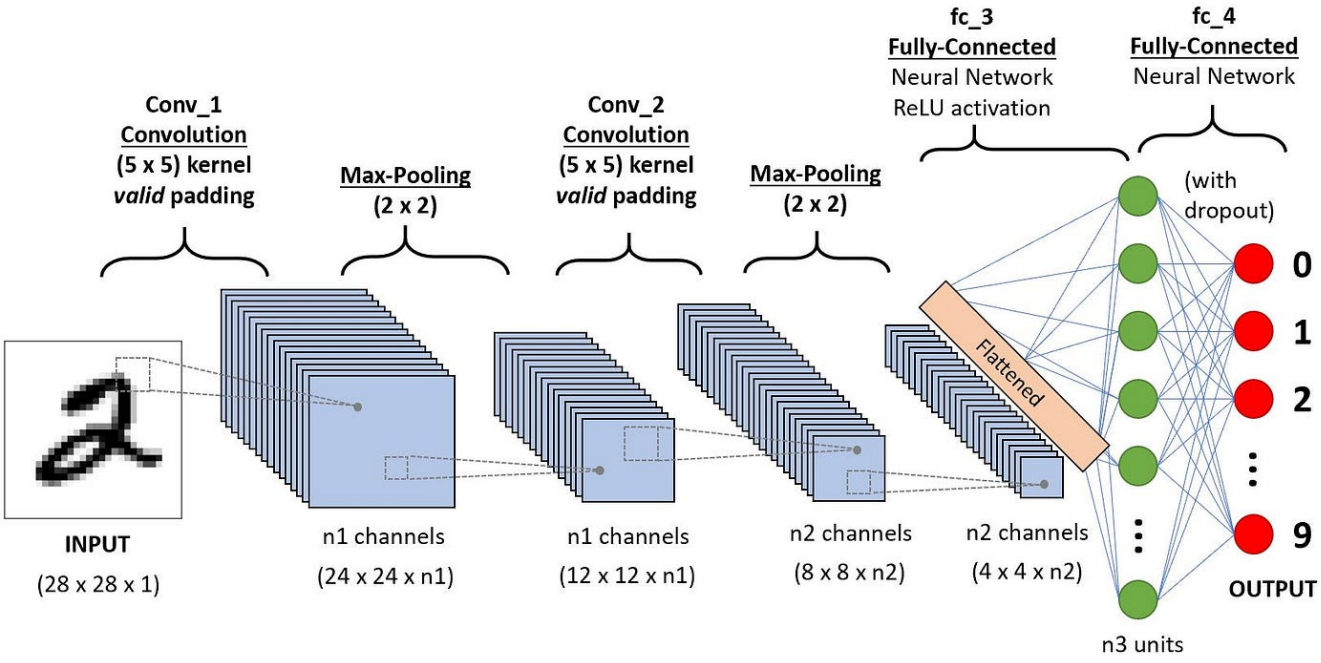


Table of Contents


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

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

Loading the dataset

Metalhead.jl is an excellent package that has a number of predefined and pretrained computer vision models. It also has a number of dataloaders that come in handy to load datasets.

```
begin
    using Statistics
    using CUDA
    using Flux, Flux.Optimise
    using MLDatasets: MNIST
    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 wh
en running on the GPU.
```

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

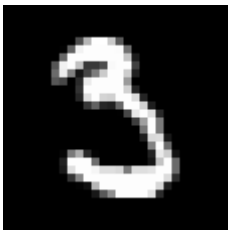
Set an environment variable to stop the system asking for approval to download data.

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

```
begin
    train_x, train_y = MNIST(split=:train)[:]
    train_x = reshape(train_x, 28, 28, 1, :)
    labels2 = onehotbatch(train_y, 0:9)
end;
```

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



```
Gray.(permutedims(reshape(train_x[:,:,:,rand(1:end)], 28, 28), (2,1)))
```

> **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).

```
begin
    train = ([(train_x[:,:,:,i], labels2[:,i]) for i in partition(1:59000, 1000)])
    |>  gpu
    valset = 59001:60000
    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).
```

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

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), 1 => 16, relu, pad=2),   # 416 parameters
       MaxPool((2, 2)),
       Conv((5, 5), 16 => 8, relu, pad=2),   # 3_208 parameters
       MaxPool((2, 2)),
       Main.var"#5#6"{typeof(reshape), typeof(size), Colon}(reshape, size, Colon()),
       Dense(392 => 120),                    # 47_160 parameters
       Dense(120 => 84),                     # 10_164 parameters
       Dense(84 => 10),                      # 850 parameters
       NNlib.softmax,
     )                   # Total: 10 arrays, 61_798 parameters, 242.633 KiB.
```

```
m3 = Chain(
  Conv((5,5), 1=>16, pad=(2,2), relu),
  MaxPool((2,2)),
  Conv((5,5), 16=>8, pad=(2,2), relu),
  MaxPool((2,2)),
  x -> reshape(x, :, size(x, 4)),
  Dense(392, 120),
  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.

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.

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

```
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:

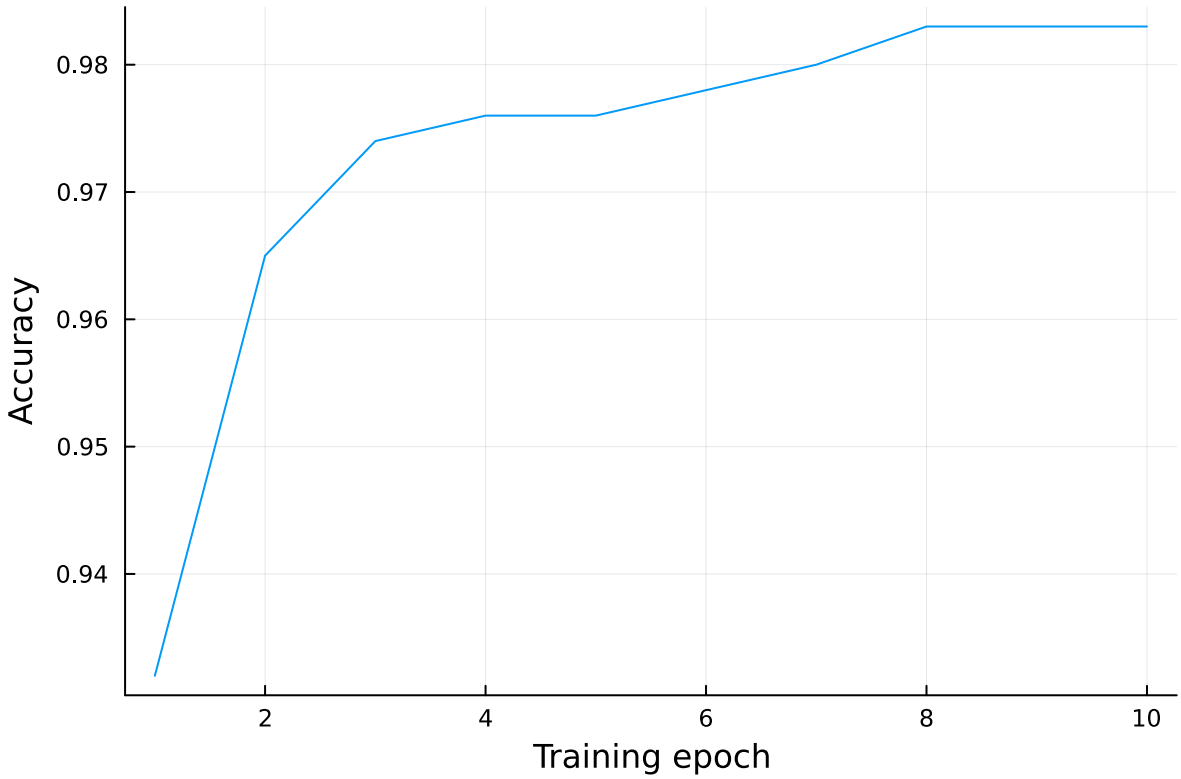
```
accuracy (generic function with 1 method)
```

```
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.

```
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.

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.

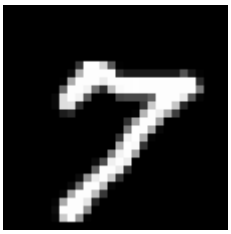
Next, display images from the test set.

```
begin
    test_x, test_y = MNIST(split=:test)[:]
    test_x = reshape(test_x, 28, 28, 1, :)
end;
```

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

```
test =
 [(28×28×1×1000 Array{Float32, 4}:                                          , 1
   [:, :, 1, 1] =
```

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



```
Gray.(permutedims(reshape(test_x[:,:,:,image_index], 28, 28), (2,1)))
```

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

```
10×5 Matrix{Float32}:
 2.6064f-5    0.000244065  1.43686f-5   7.2126f-10   0.223917
 6.13563f-14  2.24301f-8   2.37153f-10  1.30813f-9   8.05749f-13
 3.84008f-6   5.38546f-6   1.57655f-6   3.13186f-6   0.000291565
 9.6243f-8    5.244f-8     6.9135f-9    0.999979     2.31259f-8
 8.91487f-12  9.51078f-6   1.08959f-8   3.7118f-11   8.46322f-8
 1.43009f-7   8.85482f-7   6.75388f-6   1.44185f-5   5.49676f-6
 1.4831f-7    0.999697     0.99995      1.49637f-13  0.00013025
 4.83133f-8   7.4789f-12   9.43156f-13  8.69421f-9   7.68097f-5
 0.999963     4.28085f-5   2.72457f-5   2.23897f-6   0.775539
 6.48617f-6   1.53517f-8   2.82526f-11  1.69011f-6   3.93164f-5
```

```
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.

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.

```
0.982
```

```
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.

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	0.988776	0
2	0.989427	1
3	0.979651	2
4	0.973267	3
5	0.990835	4
6	0.983184	5
7	0.985386	6
8	0.974708	7
9	0.974333	8
10	0.959366	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.

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