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: CIFAR10
    using Images.ImageCore
    using Images
    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 j
ulia.
- If cuDNN is not installed, some Flux functionalities will not be available when
running on the GPU.
```

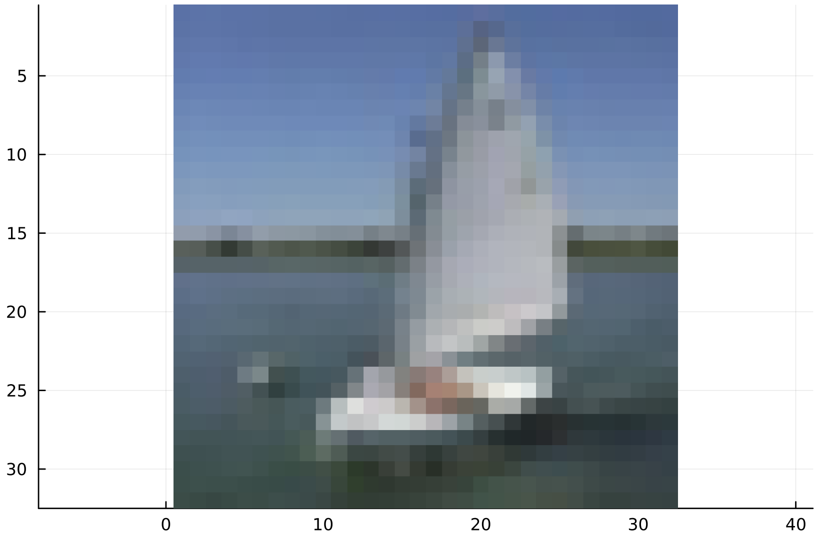
Set an environment variable to stop the system asking for approval to download data.

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

```
begin
    train_x, train_y = CIFAR10(split=:train)[:] #loadx is image data, y is label
    train_x = reshape(train_x, 32, 32, 3, :) #turn x array images to 3d array
    labels2 = onehotbatch(train_y, 0:9) #convert training labels to one hot encoded
    classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog",
    "horse", "ship", "truck"] #label array
end;
```

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




```
begin
image(x) = colorview(RGB, permutedims(x, (3, 2, 1)))
    # Display a random image from the training set
    rand_ids = rand(1:size(train_x, 4)) # Select a random index
    rand_image = train_x[:, :, :, rand_ids] # Extract the corresponding image
    plot(image(rand_image)) # Use plot function to display the image
end
```

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).

```
10×1000 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  ⋅  ⋅
```

```
begin
    train_x_small = train_x[:,:,:,1:50000]
    train_y_small = train_y[1:50000]
    #labels2 = onehotbatch(train_y_small, 0:9)

    train = ([(train_x_small[:,:,:,i], labels2[:,i]) for i in partition(1:49000,
    1000)]) |> gpu

    valset = 49001:50000
    valX = train_x_small[:,:,:,valset] |> gpu #image data for validation set
    valY = labels2[:, valset] |> gpu #label data for validation set
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.

```
m3 = Chain(
       Conv((5, 5), 3 => 16, relu, pad=2),   # 1_216 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(512 => 120),                    # 61_560 parameters
       Dense(120 => 84),                     # 10_164 parameters
       Dense(84 => 10),                      # 850 parameters
       NNlib.softmax,
     )                   # Total: 10 arrays, 76_998 parameters, 302.008 KiB.
```

```
#CNN work on batches in parralell, so 1000 images go through it simulatneosly
#change first conv layer 1 to 3, since rgb no longer greyscale
#input 32*32*3
m3 = Chain(
  Conv((5,5), 3=>16, pad=(2,2), relu), #output remain 32*32, 16 channels now 16 not 3
  MaxPool((2,2)), #reduce spatial dimension by half, 16*16 now
  Conv((5,5), 16=>8, pad=(2,2), relu), #pad is 2 so spatial 16*16, channel is 8
  MaxPool((2,2)), #half spatial dimension, so 8*8
  #Now image dimension is 8*8*8 = 512
  x -> reshape(x, :, size(x, 4)),
  Dense(512, 120),
  Dense(120, 84),
  Dense(84, 10),
  softmax) |> gpu

#First dense layer changed to take output of previous conv and pool layer
#First conv make is 16*16
#Second conv make 8*8
#La
```

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;

#m = 0.0001 - accuracy went down from 0.6950 to 0.6935
```

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:

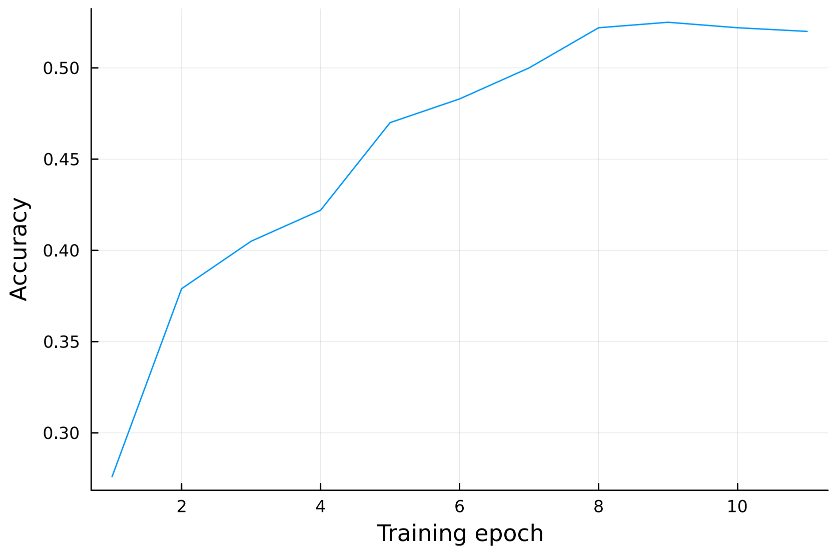
```
accuracy (generic function with 1 method)
```

```
#compare predicted label to true label, onecold convert model output to class label
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
begin
    train_acc = Float32[]
    train_epochs = Int32[]
    epochs = 11
    for epoch = 1:epochs
        #loop through each batch, so 59 iterations
        #each batch, compute loss
        for d in train
            #gradient of loss function with respect to parms
            #'do' gradient() uses, so gradient for parms with ('do') loss
            gs = gradient(Flux.params(m3)) do
                l = loss2(d...) #loss function of current batch 'd', parallelized
                #line above put batch through CNN, then compute loss
            end
            update!(opt2, Flux.params(m3), gs)
        end
        push!(train_acc, accuracy(valX, valY))
        push!(train_epochs, epoch)
    end
end
```




```julia
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.

```julia
begin
    #loaf cifar10 test set
    test_x, test_y = CIFAR10(split=:test)[:] #load x is image data, y is label
    test_x = reshape(test_x, 32, 32, 3, :) #turn x array images to 3d array
end;
```

```julia
test_labels = onehotbatch(test_y, 0:9); #one hot encode the labels
```

Created test sets, for x and y. The code snipped below just displays the images


```julia
begin
    rnd_test = rand(1:size(test_x)[4]) #pick random indexes from total data set
    plot(image(test_x[:,:,:,rnd_test]), axis=false) #plot image, no axis
    annotate!(-1.0, 1.0, text(classes[test_y[rnd_test]+1], :blue, :right, 12))
    #annotate
end
```

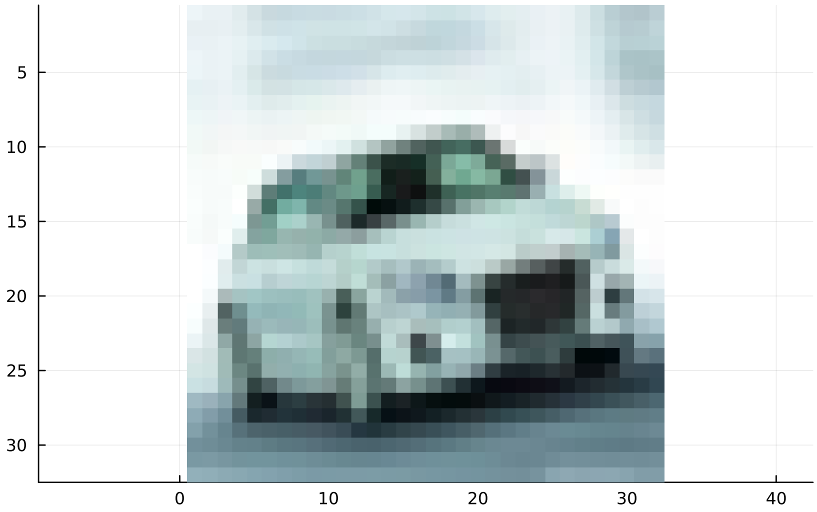
The outputs are energies for the 10 classes. Higher the energy for a class, the more the network thinks that the image is of the particular class. Every column corresponds to the output of one image, with the 10 floats in the column being the energies.

Let's see how the model fared:

```
10×20 Matrix{Float32}:
 0.0922064   0.0102547   0.0169114   …  0.00156501   0.00903615   0.0987136
 0.0363254   0.00481658  0.00421993     0.00067695   0.00117676   0.732134
 0.0317206   0.193057    0.0646506      0.00112666   0.259643     0.0489655
 0.0107373   0.201034    0.0758947      0.37967      0.0940441    0.010446
 0.043479    0.0705209   0.223071       0.00164087   0.393149     0.0243906
 0.00262163  0.152112    0.0614509   …  0.601731     0.0501104    0.00513173
 0.0227949   0.193075    0.241543       0.0100709    0.188419     0.000536064
 0.0058762   0.053236    0.283027       0.0029483    0.00279208   0.000437009
 0.561102    0.0670079   0.00330761     0.000152686  0.000677557  0.00854503
 0.193136    0.0548851   0.0259244      0.000417025  0.000952252  0.0707004
```

```julia
begin
    ids = rand(1:size(test_x)[4], 20) #20 random ids
    rand_test = test_x[:,:,:,ids] #get the images for the ids
    rand_truth = test_y[ids] #true labels for selected ids
    rand_out = m3(rand_test) #input random test images into neureal network
end
```

Predicted Label: automobile




```
begin
    displayed_image = rand_test[:,:,:,inx] #get image at index of slider value
    displayed_label_index = argmax(rand_out[:, inx]) # Get the index of the predicted label
    predicted_label = classes[displayed_label_index]


    plot(image(displayed_image)) # Display image
    title!("Predicted Label: $predicted_label") #Display predicted label
end
```

```
@bind inx Slider(1:1:20, default=1)
```

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.

```
0.538
```

```
begin

    test = [(test_x[:,:,:,i], test_labels[:,i]) for i in partition(1:size(test_x, 4), 1000)]
    accuracy(test[1]...)
end
```

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]) #makes prediction on first elem of the ith batch
        lab = test[i][2] #Retrieves the true labels for the images in the batch
        for j = 1:1000 #for each element in current batch
            pred_class = findmax(preds[:, j])[2] #most predicted class for image
            actual_class = findmax(lab[:, j])[2] #actual class for image
            if pred_class == actual_class
                class_correct[pred_class] += 1 #correct prediction increment counter
            end
            class_total[actual_class] += 1 #total count
        end
    end
end
```

	accuracy	class
1	0.517	"airplane"
2	0.614	"automobile"
3	0.375	"bird"
4	0.302	"cat"
5	0.462	"deer"
6	0.512	"dog"
7	0.729	"frog"
8	0.468	"horse"
9	0.76	"ship"
10	0.5	"truck"

```
begin
    using DataFrames
    DataFrame(accuracy=(class_correct ./ class_total), class=classes)
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.