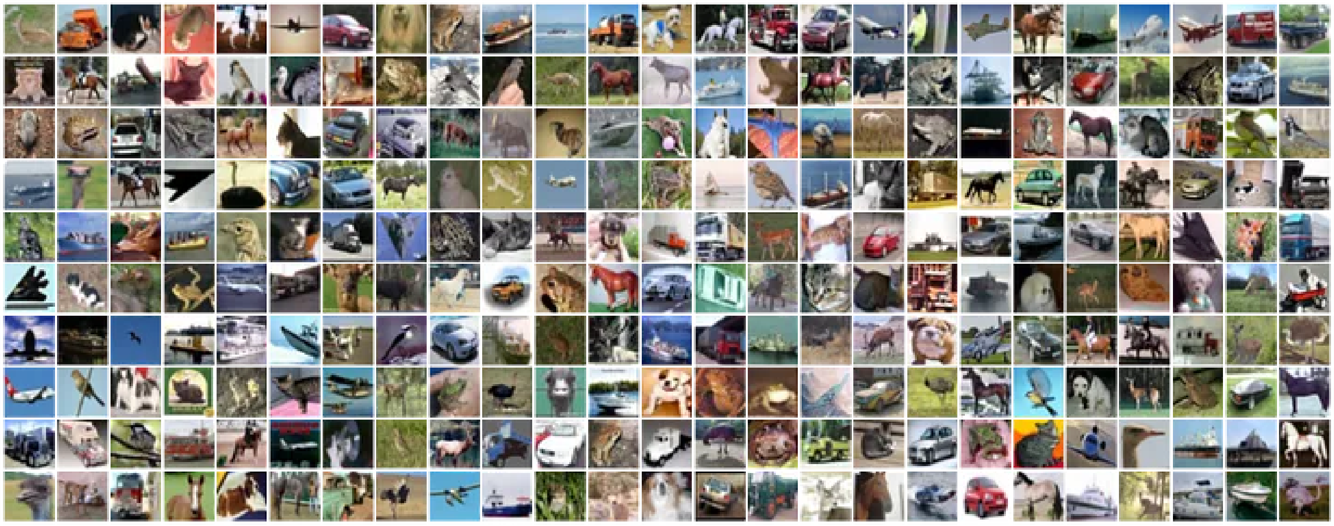
Training a colour image classifier using `Flux`

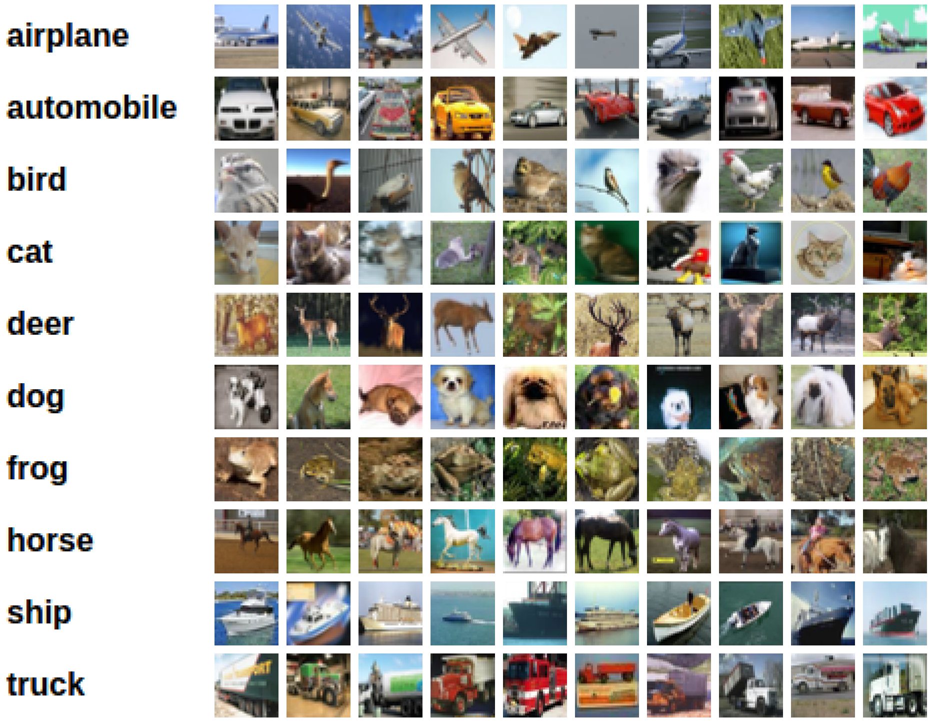


Tip

Hidden below is a useful snippet of HTML to setup a `restart` button in case training gets out of hand.

Restart

Table of Contents




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

This is a slightly more complex learning task than the MNIST example. CIFAR10 is a dataset of 50k tiny coloured training images split into 10 classes.

You need to do the following steps in order:

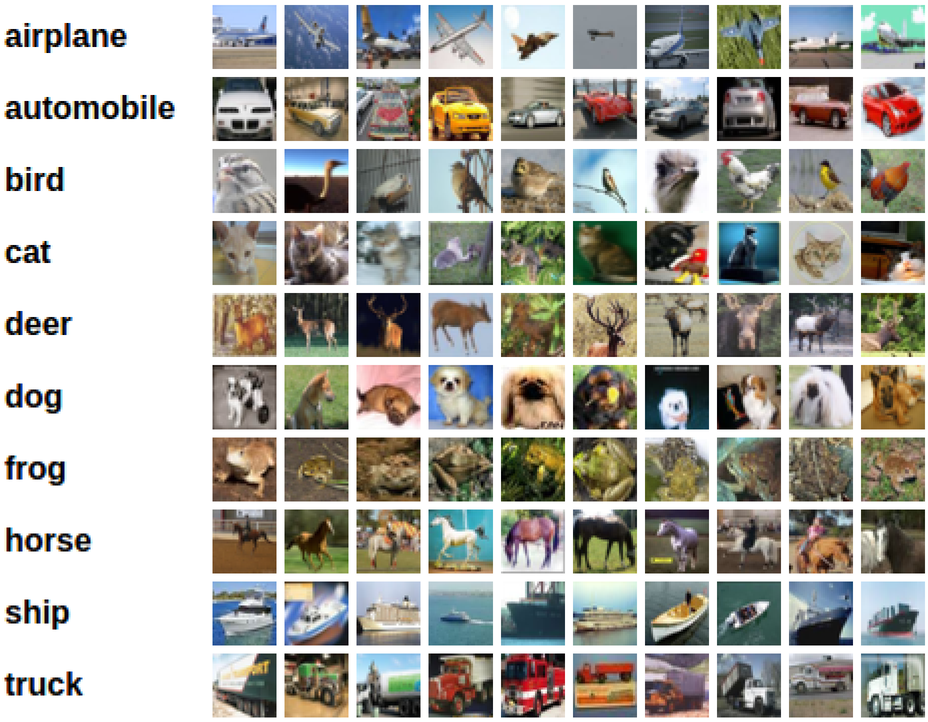
- Load CIFAR10 training and test datasets
- Define a Convolution Neural Network
- Define a loss function
- Train the network on the training data

- Test the network on the test data

Again, most of the steps are identical with what we did for MNIST task, but some dimesnsion adjustments are required because the images are slightly bigger and also involve three colour channels.

Load the dataset

The image gives an idea of the range of images in each of the 10 categories.



Again, we'll get the data from the MLDatasets repository.

```
begin
    using Statistics
    using Flux, Flux.Optimise
    using MLDatasets: CIFAR10
    using Images.ImageCore
    using Flux: onehotbatch, onecold
    using Base.Iterators: partition
    using MLUtils
    using Plots
    using Logging
    using DataFrames
    using Images
end
```

```
begin
using Random
ENV["DATADEPS_ALWAYS_ACCEPT"] = true
Random.seed!(31415927) # for reproducibility
#gpu = Flux.get_device(; verbose=true)
end;

```

```
begin
train_x, train_y = CIFAR10(split=:train)[:]
train_labels = onehotbatch(train_y, 0:9)
classes = ["airplane", "automobile", "bird", "cat",
"deer", "dog", "frog", "horse", "ship", "truck"]
end;

```

The images are simply 32 x 32 matrices of numbers in 3 channels (R,G,B). The train_x array contains 50,000 images converted to 32 x 32 x 3 arrays with the third dimension being the 3 channels (R,G,B). Let's take a look at a random image from train_x. However, to do this we need to define a function called `image`, which calls `colorview` on the training image, which we have to permute from 32x32x3 to 3x32x32:

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

```julia
image(x) = colorview(RGB, permutedims(x, (3, 2, 1)))
```

Rehape data for training with `flux`

We can now arrange them into batches of 1,000. This process is called minibatch learning, which is a popular method of training large neural networks. Rather that sending the entire dataset at once, we break it down into smaller chunks (called minibatches) that are typically chosen at random, and train only on them.

The first 49k images (in batches of 1,000) will be our training set, and the rest is for validation. `partition` handily breaks down the set we give it into consecutive chunks (1,000 in this case).

Task 1

Partition train_x into training and validation parts, along the lines done for the MNIST example.

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

```
begin
    train_batches = [(train_x[:,:,:,i], train_labels[:,i])
        for i in partition(1:49000, 1000)]

    validation_batches = [(train_x[:,:,:,i], train_labels[:,i])
        for i in partition(49001:50000, 1000)]
end
```

Note that `train` is an array of tuples, where the first tuple element is the image and the second is the label. This is the format in which the `Flux` defined model expects its training data.

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 from which to extract features. It creates higher-order features as it goes into deeper layers, making it suitable for images, where the structure of the image will help us determine which class to which it belongs.

In this case we use two convolutional layers of 16 and 8 channels, respectively. Each convolution phase is passed through a pooling layer, which reduces the image's dimentionality. The `SamePad()` function is used to ensure appropriate padding is used to preserve the dimensions of the original image.

Finally, the 3D array is flattened to a 512 element 1D vector, which is then passed through a sequence of fully-connected layers to reduce its length to 10. Finally a `softmax` transformation is applied to the 10 element output vector to transform the outputs to probabilities.

Model fix

I neglected to use padding in the last version of the template. This resulted in the convolution not preserving the original dimensions of the image. The use of SamePad() to calculate the required padding fixes this.

```
model = Chain(
          Conv((5, 5), 3 => 16, relu, pad=2),   # 1_216 parameters
          MaxPool((2, 2)),
          Conv((5, 5), 16 => 16, relu, pad=2),  # 6_416 parameters
          MaxPool((2, 2)),
          Conv((5, 5), 16 => 8, relu, pad=2),   # 3_208 parameters
          MaxPool((2, 2)),
          Flux.flatten,
          Dense(128 => 256),                    # 33_024 parameters
          Dense(256 => 128),                    # 32_896 parameters
          Dense(128 => 10),                     # 1_290 parameters
          softmax,
        )                   # Total: 12 arrays, 78_050 parameters, 305.938 KiB.
```

```
model = Chain(
    Conv((5,5), 3=>16, pad=SamePad(), relu),
    MaxPool((2,2)),
    Conv((5,5), 16=>16, pad=SamePad(), relu), #New Conv Layer
    MaxPool((2,2)), #New Pooling layer
    Conv((5,5), 16=>8, pad=SamePad(), relu),
    MaxPool((2,2)),
    Flux.flatten,
    Dense(128, 256),
    Dense(256, 128),
    Dense(128, 10),
    softmax)
```

Baseline Model Time

10 Epochs: Accuracy = 0.515, Time = 1079s

Part (a) layer addition

10 Epochs: Accuracy = 0.484, Time = 1189s (Potenital Problems here)

Part (b) layer addition

10 Epochs: Accuracy = 0.55, Time = 1406s

Differences

The baseline model serves as a reference point for comparison. Its accuracy and runtime represent the performance of the simplest architecture.

For the addition of the convolutional and pooling layer, we observe an increase in training time. This can be attributed to the additional computation required for the forward and backward passes. The decrease in accuracy, however, may be due to the lack of added dense layers, which were introduced in Part (b).

The increase in accuracy seen with the addition of the dense layer, in combination with the new convolutional and pooling layers, can be explained by the model's enhanced capacity to learn global patterns and reduce redundant information. The increase in training time is expected given the additional complexity. The significant increase in runtime also suggests that dense layers are potentially more computationally intensive.

I reduced the number of epochs down to 10 to make the length of training time shorter given that I dont have access to a powerful machine.

Task 2

Make modifications to the network architecture above to (a) insert a new pair of convolutional and pooling layers between the existing 1st and 2nd ones. Use 16 filters for the new kernel; (b) insert a new `Dense` layer just before the final one that goes from a width of 256 down to 128. Modify the final `Dense` layer approprately.

Do these modifications separately and in each case calculate the training time and classification accuracy. Note that each training test may take up to 30 minutes, depending on your machine.

Comment on and explain what differences, if any, there are between the baseline model and these two modifications.

Test network

Use this partial network to check the dimension of outputs from each layer (use # to comment out layers not of interest).

```
(128, 1)
```

```
with_terminal() do
    # Test the model up to flattening step
    x = rand(Float32, 32, 32, 3, 1)  # Example input of shape 32x32x3 (one image)
    model = Chain(
        Conv((5,5), 3=>16, pad=SamePad(), relu),
        MaxPool((2,2)),
        Conv((5,5), 16=>16, pad=SamePad(), relu),
        MaxPool((2,2)),
        Conv((5,5), 16=>8, pad=SamePad(), relu),
        MaxPool((2,2)),
        Flux.flatten
    )

    output = model(x)
    println(size(output))
end
```

We will use a crossentropy loss and the Momentum optimiser here. Crossentropy is a good option when 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 adaptivity in our optimisation, preventing overshooting of the error minimum.

```
begin
    using Flux: crossentropy, Momentum
    loss(x, y) = sum(crossentropy(model(x), y))
    optimiser = 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(model(x), 0:9) .== onecold(y, 0:9))

```

Training

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.

Error message

```
InterruptException:
```

```
with_terminal() do
    correct = []
    epochs = 10 #reduced number of epochs
    for epoch = 1:epochs
        for d in train_batches
            gradients = gradient(Flux.params(model)) do
                l = loss(d...)
            end
            update!(optimiser, Flux.params(model), gradients)
        end
        acc = accuracy(validation_batches[1]...) # Using validation_batches for accuracy
        push!(correct, acc)
        println(acc)
    end
    plot(correct, ylim=(0.0, 0.75),
        legend=:none, title="Accuracy", xlabel="epoch", ylabel="proportion correct")
end
```

Testing the network

We have trained the network for 100 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 for the test test.

We need to perform the exact same preprocessing on this set, as we did on our training set.

Task 3

Partition the test set similarly to the training set.

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

```julia
begin
     test_x, test_y = CIFAR10(split=:test)[:]     # 10,000 samples for testing
     test_labels = onehotbatch(test_y, 0:9)

    test_batches = [(test_x[:,:,:,i], test_labels[:,i])
    for i in partition(1:10000, 1000)]  # Partition test set (10,000 samples)
end
```

Task 4

Test a random sample of 10 test images. Display a dataframe of outputs as below. Use a slider to display each image and its predicted class.

The dataframe below contains probabilities for the 10 classes (left column). The model's predictions are indicated by the column names.

	Classes_Actual	Image 1 (ship)	Image 2 (truck)	Image 3 (airplane)	Image 4 (cat)	Image 5 (dog)	Image 6 (deer)	Image 7 (bird)	Image 8 (ship)	Im (s
1	:airplane	0.04	0.12	0.31	0.06	0.06	0.01	0.01	0.27	0.
2	:automobile	0.47	0.14	0.09	0.11	0.03	0.02	0.01	0.03	0.
3	:bird	0.01	0.02	0.01	0.06	0.13	0.07	0.1	0.03	0.
4	:cat	0.03	0.03	0.01	0.08	0.15	0.12	0.15	0.01	0.
5	:deer	0.01	0.01	0.01	0.06	0.14	0.18	0.19	0.01	0.
6	:dog	0.02	0.03	0.0	0.08	0.22	0.13	0.11	0.01	0.
7	:frog	0.0	0.0	0.0	0.08	0.08	0.22	0.25	0.0	0.
8	:horse	0.03	0.02	0.02	0.1	0.15	0.24	0.17	0.01	0.
9	:ship	0.14	0.43	0.37	0.1	0.03	0.0	0.0	0.56	0.
10	:truck	0.24	0.22	0.18	0.28	0.02	0.02	0.0	0.08	0.

```julia
begin
    using ColorTypes

    sample_indices = rand(1:size(test_x, 4), 10)
    rand_test = test_x[:, :, :, sample_indices]
    rand_label = test_y[sample_indices]

    # Model predictions
    predictions = model(rand_test)
    predictions_rounded = round.(predictions, digits=2)

    # Create column names with image numbers and true labels
    column_names = ["Image $i ($(classes[rand_label[i] + 1]))" for i in
    1:length(rand_label)]

    # Add row names (true classes) and construct the DataFrame
    df = DataFrame(hcat(Symbol.(classes), predictions_rounded),
                   [:Classes_Actual; Symbol.(column_names)...])


    function display_image(x)
        img = colorview(RGB, permutedims(x, (3, 2, 1)))
        Images.display(img)
    end


    # Display the selected image with the corresponding accuracy
    selected_image = rand_test[:, :, :, inx]
    selected_label = rand_label[inx]
    selected_prediction = predictions_rounded[inx]

    # Display the image and its accuracy
    display_image(selected_image)
```

```
    println("True label: $(classes[selected_label + 1])")
    println("Predicted: $(selected_prediction)")

    df
```

```
32×32 reinterpret(reshape, ColorTypes.RGB{Float32}, ::Array{Float32, 3}) wi
th eltype ColorTypes.RGB{Float32}:
 RGB{Float32}(0.0666667,0.0823529,0.0823529)  …  RGB{Float32}(0.670588,0.7490
2,0.796078)
 RGB{Float32}(0.054902,0.0705882,0.0784314)      RGB{Float32}(0.384314,0.45490
2,0.411765)
 RGB{Float32}(0.0666667,0.0705882,0.0941176)     RGB{Float32}(0.286275,0.33333
3,0.231373)
 RGB{Float32}(0.12549,0.121569,0.141176)         RGB{Float32}(0.254902,0.29019
6,0.203922)
 RGB{Float32}(0.145098,0.141176,0.160784)        RGB{Float32}(0.223529,0.25882
4,0.188235)
 RGB{Float32}(0.0666667,0.0627451,0.0823529)  …  RGB{Float32}(0.262745,0.29803
9,0.211765)
 RGB{Float32}(0.054902,0.0509804,0.0705882)      RGB{Float32}(0.32549,0.36078
4,0.25098)
 ⋮                                            ⋱
 RGB{Float32}(0.854902,0.843137,0.862745)        RGB{Float32}(0.756863,0.75294
1,0.772549)
 RGB{Float32}(0.47451,0.470588,0.501961)         RGB{Float32}(0.494118,0.49019
6,0.521569)
 RGB{Float32}(0.372549,0.380392,0.419608)        RGB{Float32}(0.258824,0.2509
8,0.298039)
 RGB{Float32}(0.737255,0.74902,0.768627)         RGB{Float32}(0.247059,0.24313
7,0.270588)
 RGB{Float32}(0.854902,0.854902,0.870588)     …  RGB{Float32}(0.54902,0.54902,
0.560784)
 RGB{Float32}(0.843137,0.831373,0.854902)        RGB{Float32}(0.572549,0.56862
7,0.607843)
True label: ship
Predicted: 0.04
```

Could not get the image to actually display!!! :(

Tip

Here's some of the code needed to create the DataFrame:

```
DataFrame(round.(model(rand_test), digits=2),
    Symbol.(rand_label),
    makeunique=true)
```

```
@bind inx Slider(1:1:10, default=1)
```

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.

Overall accuracy

We iterate over the entire test set to calculate the overall model accuracy.

0.085

```
round(mean([accuracy(test[i]...) for i in 1:10]), digits=3)
```

This is much better than random chance set at 10% (since we only have 10 classes), and not bad at all for a small handcrafted network like ours.

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

```
begin
    class_correct = zeros(10)
    class_total = zeros(10)
    for i in 1:10
        preds = model(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	class
1	0.0	"airplane"
2	0.0	"automobile"
3	0.0	"bird"
4	0.0	"cat"
5	0.041	"deer"
6	0.0	"dog"
7	0.0	"frog"
8	0.032	"horse"
9	0.775	"ship"
10	0.0	"truck"

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

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