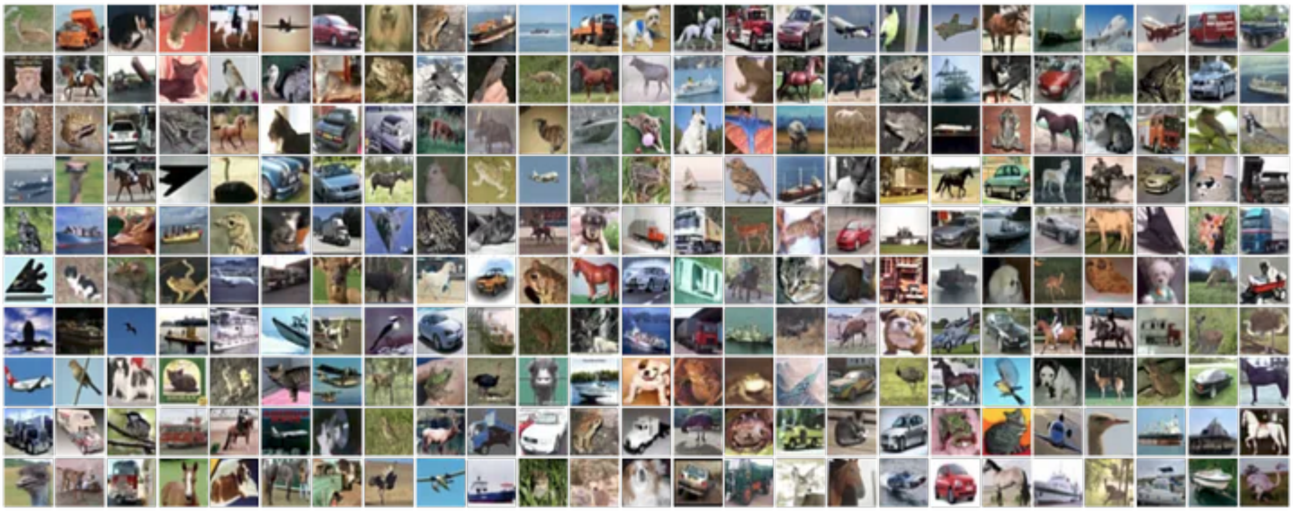
Training a colour image classifier using `Flux`



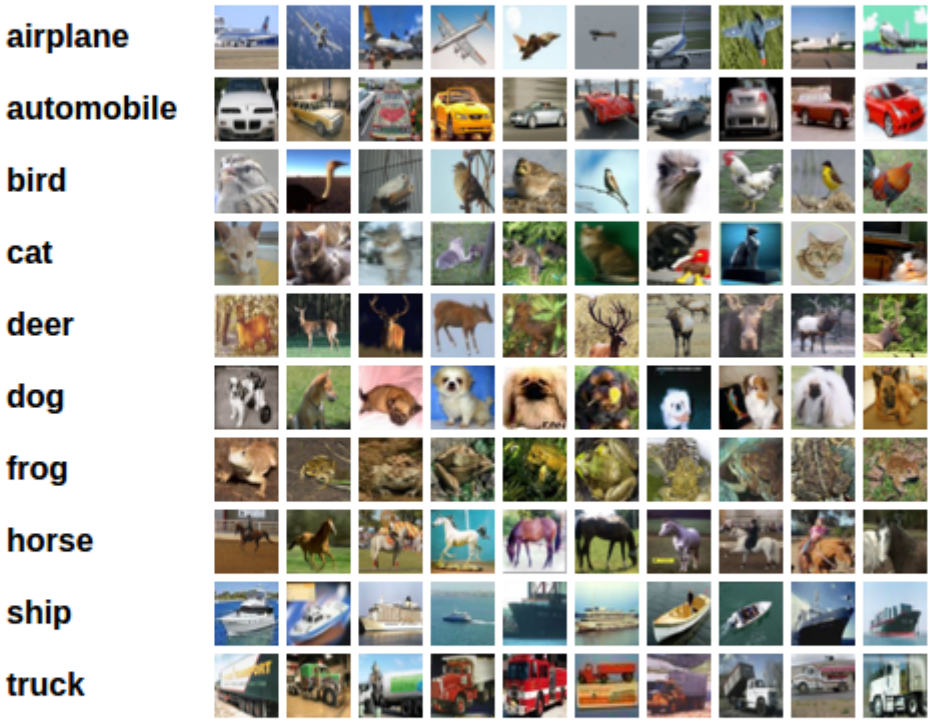
Tip

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

Restart

Table of Contents

Training a colour image classifier using `Flux`
Loading the dataset



Reshape data for training with `flux`
Task 1:
Defining the Classifier
Test network
Training
Testing the network
Overall accuracy

```julia
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:

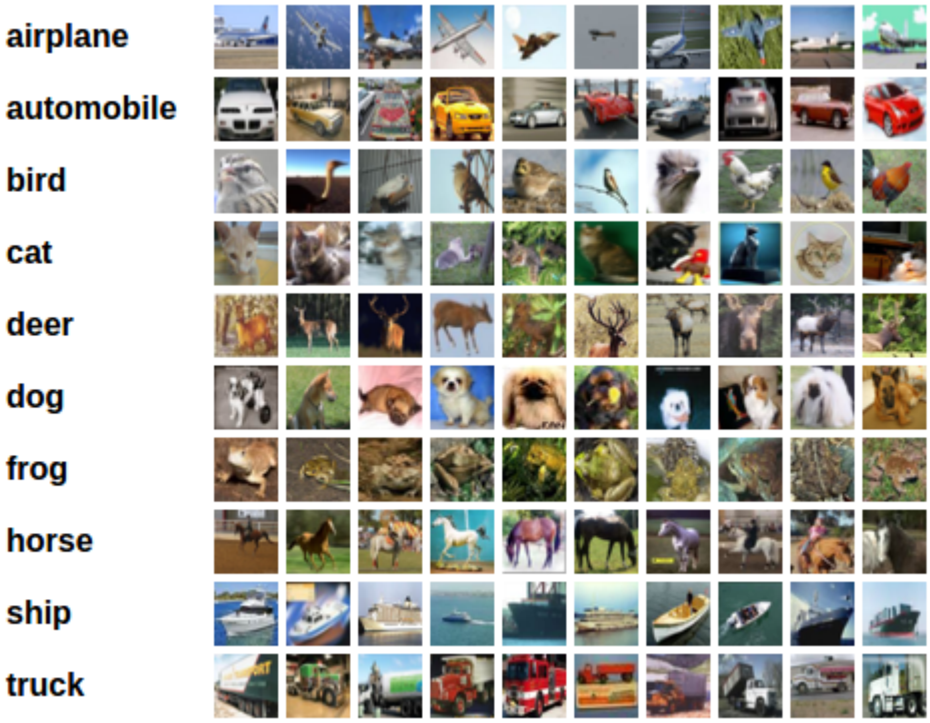
- Load CIFAR10 training and test datasets
- Define a Convolution Neural Network
- Define a loss function
- Train the network on the training data

- 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.

Loading the dataset

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



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

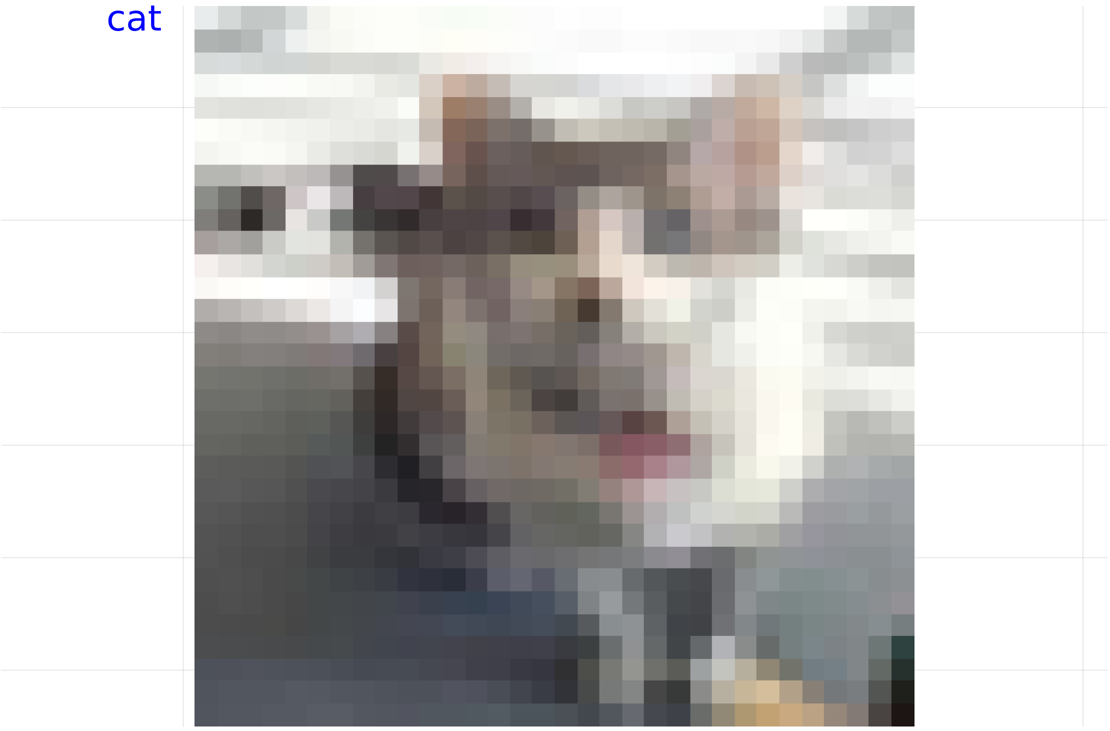
```
begin
    using Statistics
    using Flux, Flux.Optimise
    using MLDatasets: CIFAR10
    using Images.ImageCore
    using Flux: onehotbatch, onecold
    using Base.Iterators: partition
    using MLUtils
    using Plots
    using cuDNN
    using CUDA
end
```

```
begin
    ENV["DATADEPS_ALWAYS_ACCEPT"] = true
    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)))
```



```julia
begin
    rand_train = rand(1:size(train_x)[4])
    plot(image(train_x[:,:,:,rand_train]), axis=false)
    annotate!(-1.0, 1.0, text(classes[train_y[rand_train]+1], :blue, :right, 12))
end
```

Reshape data for training with `flux`

```julia
md"""
### Reshape 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 Updated

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

Task 1:

```
[(32×32×3×100 CuArray{Float32, 4, DeviceMemory}:
  [:, :, 1, 1] =
```

```
begin
    train = [(train_x[:, :, :, i], train_labels[:, i])
        for i in partition(1:49000, 100)] |> gpu

    validate = [(train_x[:, :, :, i], train_labels[:, i])
        for i in partition(49000:50000, 100)] |> gpu
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.

Finally, the 3D array is flattened to a 200 element 1D vector, which is then passed through a sequence of fully-connected layers to reduce its length from 200 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 => 8, relu, pad=2),   # 3_208 parameters
          MaxPool((2, 2)),
          Flux.flatten,
          Dense(512 => 256),                    # 131_328 parameters
          Dense(256 => 10),                     # 2_570 parameters
          softmax,
        )                   # Total: 8 arrays, 138_322 parameters, 1.555 KiB.
```

```julia
model = Chain(
    Conv((5,5), 3=>16, pad=SamePad(), relu),
    MaxPool((2,2)),
    Conv((5,5), 16=>8, pad=SamePad(), relu),
    MaxPool((2,2)),
    Flux.flatten,
    Dense(512, 256),
    Dense(256, 10),
    softmax) |> gpu
```

```
model_a = Chain(
            Conv((5, 5), 3 => 16, relu, pad=2),   # 1_216 parameters
            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(512 => 256),                    # 131_328 parameters
            Dense(256 => 10),                     # 2_570 parameters
            softmax,
          )                   # Total: 10 arrays, 144_738 parameters, 1.945 KiB.
```

```julia
model_a = Chain(
    Conv((5,5), 3=>16, pad=SamePad(), relu),
    Conv((5,5), 16=>16, pad=SamePad(), relu),
    MaxPool((2,2)),
    Conv((5,5), 16=>8, pad=SamePad(), relu),
    MaxPool((2,2)),
    Flux.flatten,
    Dense(512, 256),
    Dense(256, 10),
    softmax
) |> gpu
```

```
model_b = 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)),
            Flux.flatten,
            Dense(512 => 256),                    # 131_328 parameters
            Dense(256 => 128),                    # 32_896 parameters
            Dense(128 => 10),                     # 1_290 parameters
            softmax,
          )                   # Total: 10 arrays, 169_938 parameters, 1.859 KiB.
```

```julia
# Model_b
model_b = Chain(
    Conv((5,5), 3=>16, pad=SamePad(), relu),
    MaxPool((2,2)),
    Conv((5,5), 16=>8, pad=SamePad(), relu),
    MaxPool((2,2)),
    Flux.flatten,
    Dense(512, 256),
    Dense(256, 128),
    Dense(128, 10),
    softmax) |> gpu
```

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).

```
Output size after flattening: (128, 1)
```

```julia
with_terminal() do
    x = rand(Float32, 32, 32, 3, 1)  # Example input
    model = Chain(
        Conv((5,5), 3=>16, pad=SamePad(), relu),
        MaxPool((2,2)),
        Conv((3,3), 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("Output size after flattening: ", 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.

```julia
begin
    using Flux: crossentropy, Momentum

    # Baseline Model
    loss(x, y) = sum(crossentropy(model(x), y))
    optimiser = Momentum(0.01)

    # Model A
    loss_a(x, y) = sum(crossentropy(model_a(x), y))
    optimiser_a = Momentum(0.01)

    # Model B
    loss_b(x, y) = sum(crossentropy(model_b(x), y))
    optimiser_b = 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)
```

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

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

```julia
accuracy_a(x, y) = mean(onecold(model_a(x), 0:9) .== onecold(y, 0:9))
```

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

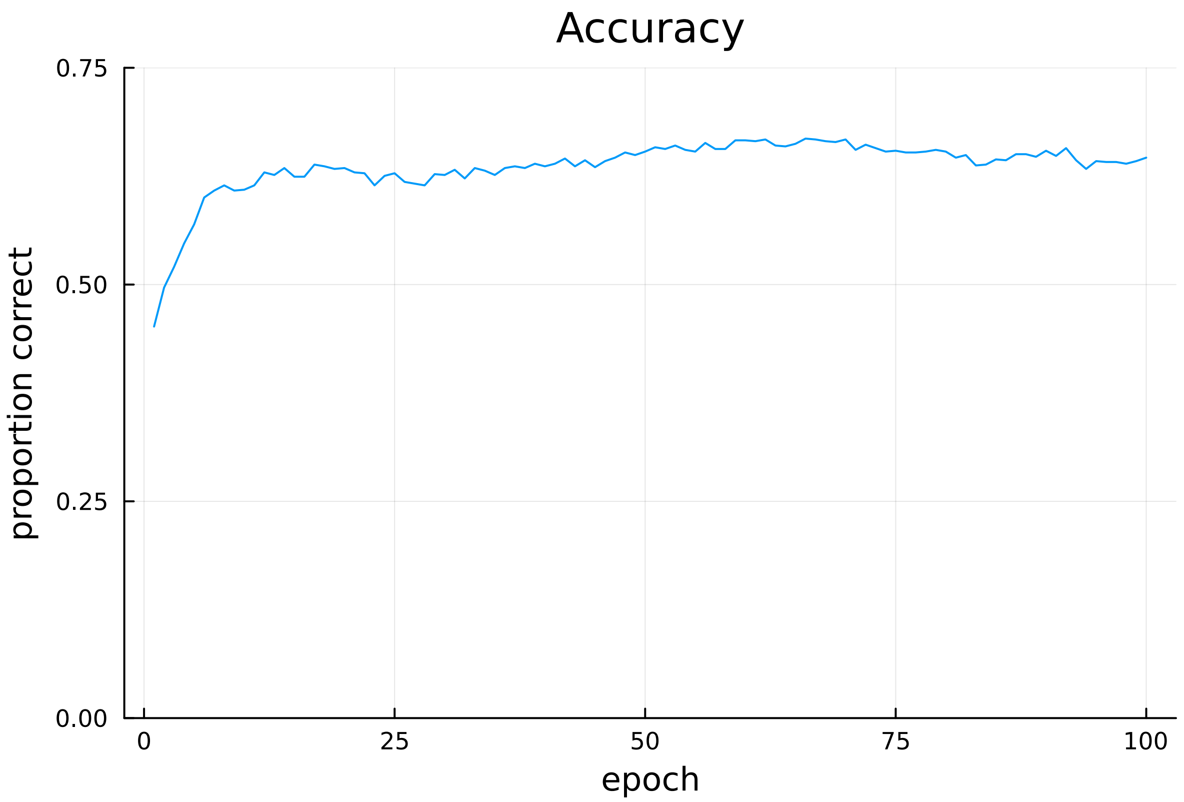
```julia
accuracy_b(x, y) = mean(onecold(model_b(x), 0:9) .== onecold(y, 0:9))
```

```
10×1001 OneHotMatrix(::CuArray{UInt32, 1, CUDA.DeviceMemory}) 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
    validate_x = cat([batch[1] for batch in validate]..., dims=4)
    validate_y = hcat([batch[2] for batch in validate]...)
end
```

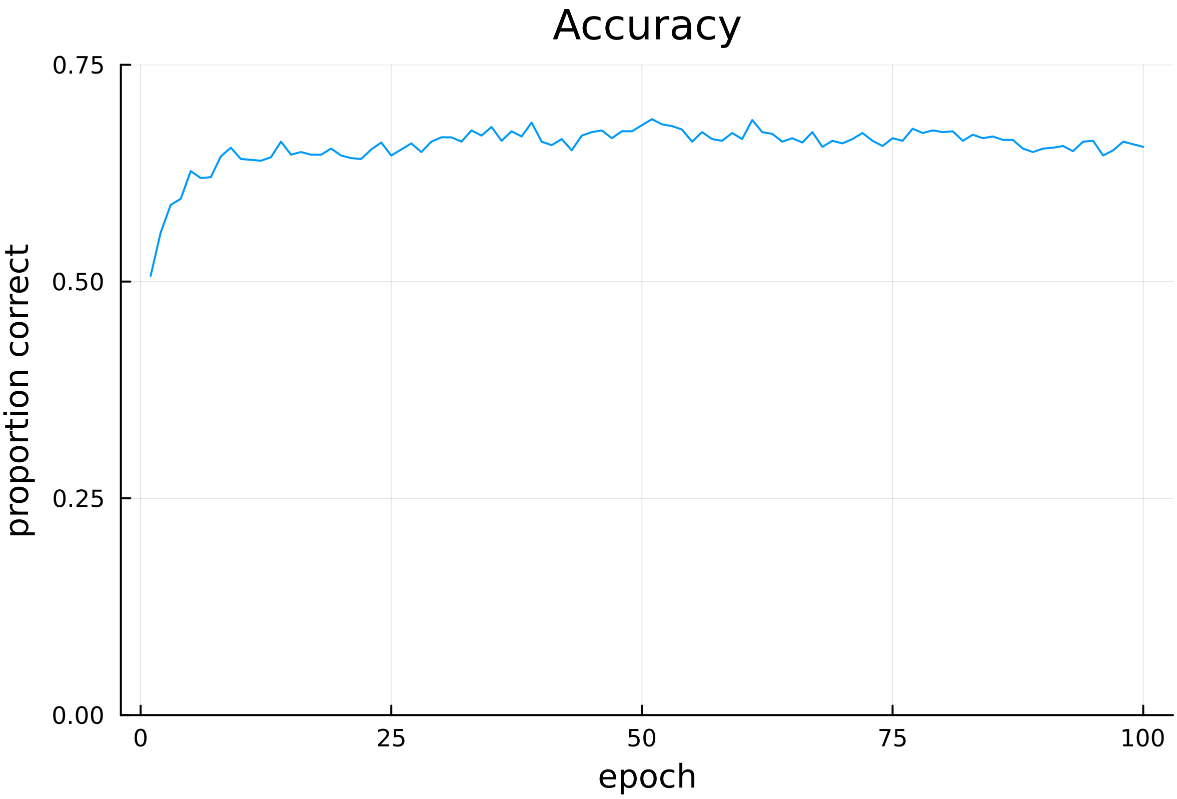
Training

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.



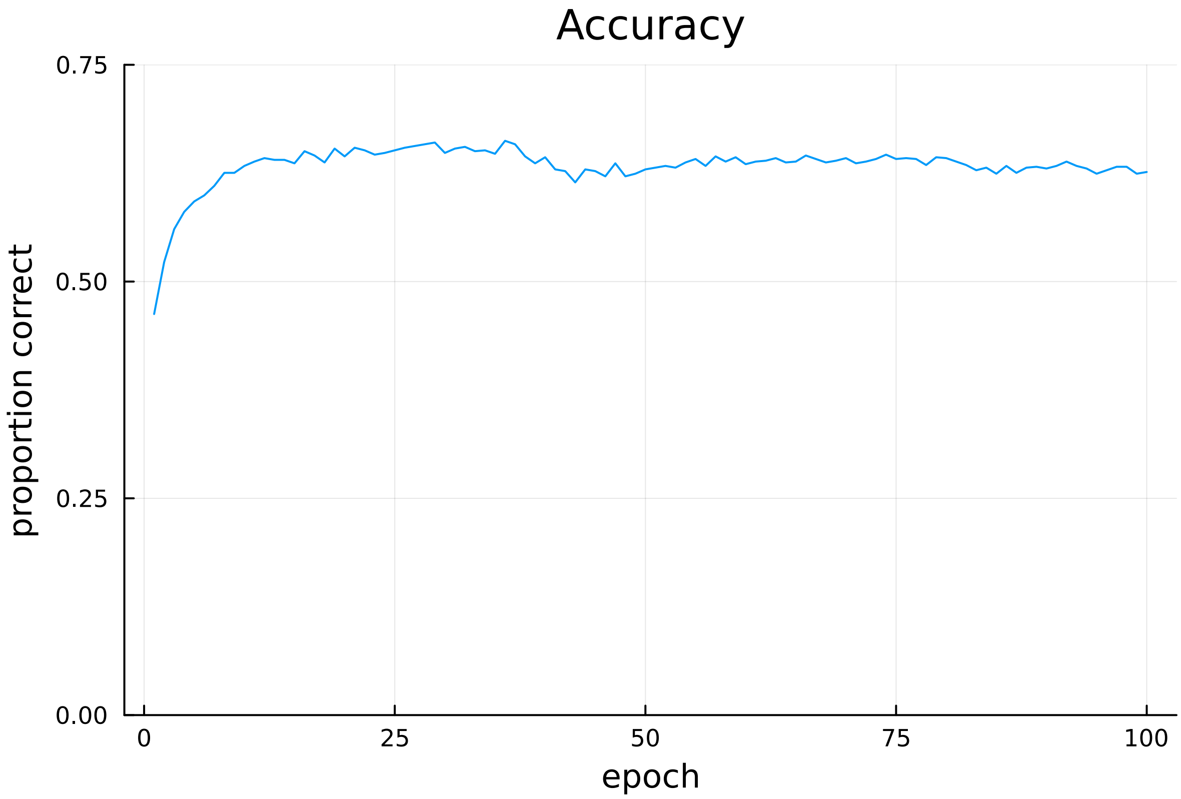
```
Standard Model - Epoch 1 Accuracy: 0.4515484515484515
Standard Model - Epoch 2 Accuracy: 0.4965034965034965
Standard Model - Epoch 3 Accuracy: 0.5204795204795205
Standard Model - Epoch 4 Accuracy: 0.5474525474525475
Standard Model - Epoch 5 Accuracy: 0.5694305694305695
Standard Model - Epoch 6 Accuracy: 0.6003996003996004
Standard Model - Epoch 7 Accuracy: 0.6083916083916084
Standard Model - Epoch 8 Accuracy: 0.6143856143856143
Standard Model - Epoch 9 Accuracy: 0.6083916083916084
Standard Model - Epoch 10 Accuracy: 0.6093906093906094
Standard Model - Epoch 11 Accuracy: 0.6143856143856143
Standard Model - Epoch 12 Accuracy: 0.6293706293706294
Standard Model - Epoch 13 Accuracy: 0.6263736263736264
Standard Model - Epoch 14 Accuracy: 0.6343656343656343
Standard Model - Epoch 15 Accuracy: 0.6243756243756243
Standard Model - Epoch 16 Accuracy: 0.6243756243756243
Standard Model - Epoch 17 Accuracy: 0.6383616383616384
Standard Model - Epoch 18 Accuracy: 0.6363636363636364
Standard Model - Epoch 19 Accuracy: 0.6333666333666333
```

```julia
with_terminal() do
    correct = []
    epochs = 100
    for epoch = 1:epochs
        for d in train
            gradients = gradient(Flux.params(model)) do
                l = loss(d...)
            end
            update!(optimiser, Flux.params(model), gradients)
        end
        acc = accuracy(validate_x, validate_y)
        push!(correct, acc)
        println("Standard Model - Epoch $epoch Accuracy: $acc")
    end
    plot(correct, ylim=(0.0, 0.75),
        legend=:none, title="Accuracy", xlabel="epoch", ylabel="proportion
correct")
end
```



```
Model_a - Epoch 1 Accuracy: 0.5064935064935064
Model_a - Epoch 2 Accuracy: 0.5564435564435565
Model_a - Epoch 3 Accuracy: 0.5884115884115884
Model_a - Epoch 4 Accuracy: 0.5954045954045954
Model_a - Epoch 5 Accuracy: 0.6273726273726273
Model_a - Epoch 6 Accuracy: 0.6193806193806194
Model_a - Epoch 7 Accuracy: 0.6203796203796204
Model_a - Epoch 8 Accuracy: 0.6443556443556444
Model_a - Epoch 9 Accuracy: 0.6543456543456544
Model_a - Epoch 10 Accuracy: 0.6413586413586414
Model_a - Epoch 11 Accuracy: 0.6403596403596403
Model_a - Epoch 12 Accuracy: 0.6393606393606394
Model_a - Epoch 13 Accuracy: 0.6433566433566433
Model_a - Epoch 14 Accuracy: 0.6613386613386614
Model_a - Epoch 15 Accuracy: 0.6463536463536463
Model_a - Epoch 16 Accuracy: 0.6493506493506493
Model_a - Epoch 17 Accuracy: 0.6463536463536463
Model_a - Epoch 18 Accuracy: 0.6463536463536463
Model_a - Epoch 19 Accuracy: 0.6533466533466533
```

```julia
with_terminal() do
    correct_a = []
    epochs = 100
    for epoch = 1:epochs
        for d in train
            gradients = gradient(Flux.params(model_a)) do
                l = loss_a(d...)
            end
            update!(optimiser_a, Flux.params(model_a), gradients)
        end
        acc = accuracy_a(validate_x, validate_y)
        push!(correct_a, acc)
        println("Model_a - Epoch $epoch Accuracy: $acc")
    end
    plot(correct_a, ylim=(0.0, 0.75),
        legend=:none, title="Accuracy", xlabel="epoch", ylabel="proportion
correct")
end
```



```
Model_b - Epoch 1 Accuracy: 0.46253746253746253
Model_b - Epoch 2 Accuracy: 0.5224775224775224
Model_b - Epoch 3 Accuracy: 0.5604395604395604
Model_b - Epoch 4 Accuracy: 0.5804195804195804
Model_b - Epoch 5 Accuracy: 0.5924075924075924
Model_b - Epoch 6 Accuracy: 0.5994005994005994
Model_b - Epoch 7 Accuracy: 0.6103896103896104
Model_b - Epoch 8 Accuracy: 0.6253746253746254
Model_b - Epoch 9 Accuracy: 0.6253746253746254
Model_b - Epoch 10 Accuracy: 0.6333666333666333
Model_b - Epoch 11 Accuracy: 0.6383616383616384
Model_b - Epoch 12 Accuracy: 0.6423576423576424
Model_b - Epoch 13 Accuracy: 0.6403596403596403
Model_b - Epoch 14 Accuracy: 0.6403596403596403
Model_b - Epoch 15 Accuracy: 0.6363636363636364
Model_b - Epoch 16 Accuracy: 0.6503496503496503
Model_b - Epoch 17 Accuracy: 0.6453546453546454
Model_b - Epoch 18 Accuracy: 0.6373626373626373
Model_b - Epoch 19 Accuracy: 0.6533466533466533
```

```julia
with_terminal() do
    correct_b = []
    epochs = 100
    for epoch = 1:epochs
        for d in train
            gradients = gradient(Flux.params(model_b)) do
                l = loss_b(d...)
            end
            update!(optimiser_b, Flux.params(model_b), gradients)
        end
        acc = accuracy_b(validate_x, validate_y)
        push!(correct_b, acc)
        println("Model_b - Epoch $epoch Accuracy: $acc")
    end
    plot(correct_b, ylim=(0.0, 0.75),
        legend=:none, title="Accuracy", xlabel="epoch", ylabel="proportion correct")
end
```

Task 2: Comments

Based on first time running*

Standard model:

- Training Time: 327 seconds
- Final Accuracy: ~64.44% (Epoch 100)

Model A:

- Training Time: 638 seconds
- Final Accuracy: ~65.53% (Epoch 100)

Model B:

- Training Time: 355 seconds
- Final Accuracy: ~63.84% (Epoch 100)

<br>

- The baseline model shows steady improvement in accuracy during the first 30–40 epochs, peaking around 64%.
- Model A achieved slightly better accuracy than the baseline model took a much longer time with the addition of a convolutional and pooling layer
- In Model A, The accuracy plateaus after epoch 50, indicating that the model has reached its learning capacity.
- Model B performed similiarly to the Baseline, so the additional dense layer wasn't significant.

Model	Training Time (s)	Final Accuracy (%)	Peak Accuracy (%)	Observations

Baseline	327	64.44	64.93	Stable performance, fastest training time.
Model A	638	65.53	66.33	Improved accuracy but slower due to added complexity.
Model B	355	63.84	65.53	Accuracy slightly the same; minimal impact.

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.

```
begin
    test_x, test_y = CIFAR10(split=:test)[:]
    test_x = Float32.(test_x)
    test_labels = onehotbatch(test_y, 0:9)

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

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

```
function test_accuracy(model, test_data)
    total_correct = 0
    total_samples = 0

    for (x, y) in test_data
        preds = model(x)
        total_correct += sum(onecold(preds, 0:9) .== onecold(y, 0:9))
        total_samples += size(x, 4)  # Number of images in the batch
    end

    accuracy = total_correct / total_samples
    return accuracy
end
```

```
begin
    # For the baseline model
    acc_baseline = test_accuracy(model, test)
    println("Baseline Model Test Accuracy: $(acc_baseline)")

    # For Model A
    acc_model_a = test_accuracy(model_a, test)
    println("Model A Test Accuracy: $(acc_model_a)")

    # For Model B
    acc_model_b = test_accuracy(model_b, test)
    println("Model B Test Accuracy: $(acc_model_b)")
end
```

```
Baseline Model Test Accuracy: 0.6378
Model A Test Accuracy: 0.6598
Model B Test Accuracy: 0.6375
```

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.

Baseline model:

Error message

> InterruptException:

```julia
begin
    using DataFrames

    rand_indices = rand(1:size(test_x, 4), 10)
    rand_test = test_x[:, :, :, rand_indices]
    rand_labels = test_y[rand_indices]

    rand_test = cu(rand_test)
    sample_preds = Array(model(rand_test))

    class_labels = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
    predicted_indices = onecold(sample_preds, 0:9)
    predicted_classes = [class_labels[i] for i in predicted_indices]

    df = DataFrame(
        "Classes/Actual" => class_labels,
        [predicted_classes[i] => round.(sample_preds[:, i], digits=2) for i in 1:10]...;
        makeunique=true
    )

    PlutoUI.HTML("""
    <div style="overflow-x: auto;">
        <table>
            <thead>
                <tr>
                    $(join(["<th>$(col)</th>" for col in names(df)], " "))
                </tr>
            </thead>
            <tbody>
                $(join(["<tr>" * join(["<td>$(df[row, col])</td>" for col in names(df)], " ") * "</tr>" for row in 1:size(df, 1)], "\n"))
            </tbody>
        </table>
    </div>
    """)
end

```

Tip

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

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

```
actual_classes =
 ["automobile", "deer", "cat", "automobile", "truck", "truck", "airplane", "frog", "air
```

```
actual_classes = [class_labels[rand_labels[i] + 1] for i in 1:10]
```

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

Predicted: truck
Actual: airplane

```julia
begin
    selected_image = rand_test[:, :, :, inx]
    selected_label_index = rand_labels[inx]
    selected_label = class_labels[selected_label_index + 1]

    selected_pred = sample_preds[:, inx]
    selected_pred_index = onecold(selected_pred, 0:9)
    selected_pred_class = class_labels[selected_pred_index + 1]

    display_image = colorview(RGB, permutedims(cpu(selected_image), (3, 1, 2)))
    plt = plot(display_image, axis=false, title="Predicted: $selected_pred_class\nActual: $selected_label")

    # display(plt)
end
```

Model a:

Classes/ Actual	automobile	deer	cat	automobile_1	truck	truck_1	airplane	frog	air
airplane	0.0	0.0	0.0	0.0	0.0	0.0	0.56	0.0	0.8
automobile	1.0	0.0	0.0	0.99	0.0	0.0	0.3	0.0	0.0
bird	0.0	0.01	0.31	0.0	0.0	0.0	0.0	0.0	0.0
cat	0.0	0.01	0.51	0.0	0.0	0.0	0.0	0.09	0.0
deer	0.0	0.71	0.03	0.0	0.0	0.0	0.0	0.01	0.0
dog	0.0	0.01	0.04	0.0	0.0	0.0	0.0	0.02	0.0
frog	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.88	0.0
horse	0.0	0.27	0.01	0.0	0.0	0.0	0.0	0.0	0.0
ship	0.0	0.0	0.0	0.0	0.0	0.0	0.14	0.0	0.0
truck	0.0	0.0	0.0	0.01	0.99	1.0	0.0	0.0	0.0

Model b:

Classes/ Actual	automobile	deer	cat	automobile_1	truck	truck_1	automobile_2	dog
airplane	0.0	0.01	0.01	0.0	0.01	0.0	0.05	0.0
automobile	0.98	0.0	0.03	1.0	0.01	0.02	0.84	0.0
bird	0.0	0.02	0.06	0.0	0.0	0.0	0.0	0.0
cat	0.0	0.01	0.51	0.0	0.0	0.0	0.0	0.29
deer	0.0	0.74	0.05	0.0	0.0	0.0	0.0	0.03
dog	0.0	0.02	0.33	0.0	0.0	0.0	0.0	0.49
frog	0.0	0.0	0.02	0.0	0.0	0.0	0.0	0.16
horse	0.0	0.18	0.02	0.0	0.0	0.0	0.0	0.03
ship	0.0	0.0	0.0	0.0	0.0	0.0	0.03	0.0
truck	0.02	0.0	0.0	0.0	0.98	0.98	0.08	0.0

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.

Baseline Model:

```julia
begin
    class_correct = zeros(10)
    class_total = zeros(10)

    for i in 1:length(test)
        # Get predictions and labels for the batch
        preds = model(test[i][1])       # preds: (10, batch_size) on GPU
        lab = test[i][2]                # lab: OneHotMatrix on GPU

        # Move predictions to CPU and get predicted class indices
        pred_classes = onecold(cpu(preds), 0:9)  # Convert to class indices

        actual_classes = Int.(cpu(lab.indices)) .- 1  # Convert UInt32 indices to Int and adjust for 0-based indexing

        # Update counts
        for c in 0:9
            idx = (actual_classes .== c)
            class_total[c + 1] += sum(idx)
            class_correct[c + 1] += sum(pred_classes[idx] .== c)
        end
    end
end

```

	accuracy	class
1	0.623	"airplane"
2	0.822	"automobile"
3	0.497	"bird"
4	0.359	"cat"
5	0.616	"deer"
6	0.571	"dog"
7	0.718	"frog"
8	0.768	"horse"
9	0.731	"ship"
10	0.673	"truck"

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

```
0.615
```

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

Model a:

```julia
begin
    class_correct_a = zeros(10)
    class_total_a = zeros(10)

    for i in 1:length(test)
        preds_a = model_a(test[i][1])
        lab = test[i][2]

        pred_classes_a = onecold(cpu(preds_a), 0:9)

        actual_classes = Int.(cpu(lab.indices)) .- 1

        for c in 0:9
            idx = (actual_classes .== c)
            class_total_a[c + 1] += sum(idx)
            class_correct_a[c + 1] += sum(pred_classes_a[idx] .== c)
        end
    end
end

```

df_model_a =

	accuracy	class
1	0.661	"airplane"
2	0.706	"automobile"
3	0.529	"bird"
4	0.459	"cat"
5	0.711	"deer"
6	0.615	"dog"
7	0.642	"frog"
8	0.705	"horse"
9	0.783	"ship"
10	0.787	"truck"

```julia
df_model_a = DataFrame(accuracy=(class_correct_a ./ class_total_a),
class=classes)
```

mean_accuracy_a = 0.66

Model b:

```julia
begin
    class_correct_b = zeros(10)
    class_total_b = zeros(10)

    for i in 1:length(test)
        preds_b = model_b(test[i][1])
        lab = test[i][2]

        pred_classes_b = onecold(cpu(preds_b), 0:9)

        actual_classes = Int.(cpu(lab.indices)) .- 1

        for c in 0:9
            idx = (actual_classes .== c)
            class_total_b[c + 1] += sum(idx)
            class_correct_b[c + 1] += sum(pred_classes_b[idx] .== c)
        end
    end
end
```

```
df_model_b =
```

	accuracy	class
1	0.661	"airplane"
2	0.685	"automobile"
3	0.496	"bird"
4	0.455	"cat"
5	0.632	"deer"
6	0.601	"dog"
7	0.704	"frog"
8	0.689	"horse"
9	0.619	"ship"
10	0.833	"truck"

[illegible]	[illegible]	truck

```
df_model_b = DataFrame(accuracy=(class_correct_b ./ class_total_b), class=classes)
```

```
mean_accuracy_b = 0.638
```