```
import torch
from torch import nn
from d2l import torch as d2l

import sys
import os

os.path.abspath(os.path.join(os.getcwd(), os.pardir))

from dog_and_cat_classifier_cnn_from_scratch.model import Conv2D, MaxPool2d, ReLU, GlobalAvgPool2d,
LinearRegression, SoftmaxRegression, SGDFromScratch, CrossEntropyError, Dropout
```

3.0 ResNet Architecture

The ResNet architecture introduces the **Residual Learning** block, a fundamental improvement on deeper models. We will talk about the motivation for a deeper neural network, the core problem that hinders it, and how ResNet mitigates this problem.

For this notebook, I'd like to quote my Multivariable Calculus professor, I-Shen Lai. Say hello to Professor Lai!

> "Haha… you can quote me if you want." I-Shen Lai, September 3rd 2025.

3.1 Deep Learning

Deep Learning models are made up of multiple sequential layers of other model, usually composed of the models we made in earlier notebooks. The addition of newer layers allow for more representation of complex features, allowing more information to be learned resulting in an even lower loss [1].

However, there is a limit to the number of layers we can effectively add. In practice, when we reach tens of layers, the model faces a **Degradation Problem** that result in stagnant accuracy, accompanied by higher training errors [1].

He et al. proposed that the layers themselves could not fit or approximate to the **identity function**, which is the function that simply return input itself [1].

So if we could somehow skip layers we deem useless or that it would just result in overfitting, then theoretically, we could improve our model, where adding additional layers would not worsen its performance. > *Drum Rolls*

3.2 Residual Learning Block

Residual Learning Block introduced by He has a property of skip connections, in which our layer could either try to learn the underlying function that we desire or the identity function.

What is this underlying function? It's the function that model reality and is the ultimate truth. Throughout our lesson, we have been using `LinearRegression` to learn if there's a linear relationship, or we add `ReLU` to model non-linear and complex ones. We define as follows: the input x, the underlying function $U(x)$, and the layer's function $F(x)$.

$$F(\mathbf{x}) = U(\mathbf{x}) - \mathbf{x}$$

Cleverly, He et al. arranged it to the following

$$U(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$$

It's explained that the skip connection is achieved when $F(\mathbf{x})$ outputs 0, thus achieving $U(\mathbf{x}) = \mathbf{x}$, which is the identity function.

3.3 Batch Normalization

Before, we work out the implementation in code. He et al. adopts batch normalization on its architecture, which helps model to converge faster in its training [2]. It must be applied in between affine function and non-linearity function.

According to D2L, Batch normalization normalizes a layer's output with the batch's mean $\hat{u}_{\mathcal{B}}$ and standard deviation $\hat{\sigma}_{\mathcal{B}}$

$$BN(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{u}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}} + \beta$$

$$\hat{u}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x}$$

$$\hat{\sigma}^2_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{u}_{\mathcal{B}})^2$$

```python
# The following definitions are a direct copy from the d2l, I'm too lazy to work this one out
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use is_grad_enabled to determine whether we are in training mode
    if not torch.is_grad_enabled():
        # In prediction mode, use mean and variance obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # In training mode, the current mean and variance are used
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = (1.0 - momentum) * moving_mean + momentum * mean
        moving_var = (1.0 - momentum) * moving_var + momentum * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.data, moving_var.data

class BatchNorm2d(d2l.Module):
    # num_features: the number of outputs for a fully connected layer or the
    # number of output channels for a convolutional layer. num_dims: 2 for a
    # fully connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # MODIFIED: ADDED REGISTER_BUFFER, NOT HAVING THIS REALLY MESSED ME UP!
        # The variables that are not model parameters are initialized to 0 and
        # 1
        self.register_buffer('moving_mean', torch.zeros(shape))
        self.register_buffer('moving_var', torch.ones(shape))

    def forward(self, X):
        # If X is not on the main memory, copy moving_mean and moving_var to
        # the device where X is located
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.1)
        return Y
```

3.4 L2 Regularization

We will also use L2 Regularization to help with overfitting. We would want to punish our model if the weight is too large, so that no one characteristics is overrepresented by our model weights. As mentioned in the first notebook, this is also known as "Weigh Decay" and it is one of the most used methods to combat overfitting.

This will goes on to the training part of our algorithm

```python
def L2Regularization(w, lambd):
    return (lambd / 2) * torch.norm(w, 2) ** 2
```

3.5 ResNet-50 Implementation

We will need to design a layer first before constructing our model. A bottleneck design will be utilized since we are going to build a ResNet-50. Each Residual Learning block will consist of a 1x1, 3x3 and 1x1 convolution layer (in that order), which is for down-sampling our features and then upscaling it to the same dimension [1]. For every Convolution layer we added, we add `BatchNorm2d` and `ReLU` for non-linearity and stability.

```python
class ResNetLayer(d2l.Module):
    def __init__(self, in_channels, out_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.ReLU = ReLU()

        self.conv1 = Conv2D(in_channels, out_channels, kernel_size=1, stride=strides)
        self.bn1 = BatchNorm2d(out_channels, 4)
        self.conv2 = Conv2D(out_channels, out_channels, kernel_size=3, padding=1, stride=1)
        self.bn2 = BatchNorm2d(out_channels, 4)
        self.conv3 = Conv2D(out_channels, out_channels * 4, kernel_size=1, stride=1)
        self.bn3 = BatchNorm2d(out_channels * 4, 4)

        # Skip connection to match input/output dimensions if needed
        if use_1x1conv:
            self.conv4 = Conv2D(in_channels, out_channels * 4, kernel_size=1, stride=strides)
        else:
            self.conv4 = None

        self.ReLU = ReLU()

    def forward(self, X):
        Y = self.ReLU(self.bn1(self.conv1(X)))
        Y = self.ReLU(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))

        # add skip connection
        if self.conv4:
            X = self.conv4(X)
        Y += X

        return self.ReLU(Y)
```

With the individual layers implemented. We can start building our ResNet-50 model. For smaller ResNet models though, we would use a normal design with just two Convolution layers.

A modification is required to prevent overfitting. We will add three `Dropout` functions: two after pooling, and one after our `LinearRegression` layer. Since both the pooling functions, and the `LinearRegression` module are feature aggregators, we would want our model to randomly activate or drop certain neurons to see if it translates to a better performing model on the validation data, to reduce the tendency of a complex model to 'memorize' the data.

The ResNet-50 is implemented below. Marvel at how simple it looks! (or not, if you have been following our notebooks series)

```
class ResNet50(d2l.Classifier):
    def __init__(self, num_classes, lr,  in_channels=1, dropout_rate=0.5):
        super().__init__()
        self.lr = lr
        self.bias = True
        self.dropout_rate = dropout_rate
        self.conv1 = Conv2D(kernel_size=7, in_channels=in_channels, out_channels=64, stride=2, padding=3)
        self.bn1 = BatchNorm2d(64, 4)
        self.relu = ReLU()
        self.pool1 = MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.dropout1 = Dropout(p=dropout_rate)
        self.conv2 = nn.Sequential(
            ResNetLayer(in_channels=64, out_channels=64, use_1x1conv=True, strides=1),
            ResNetLayer(in_channels=256, out_channels=64),
            ResNetLayer(in_channels=256, out_channels=64)
        )
        self.conv3 = nn.Sequential(
            ResNetLayer(in_channels=256, out_channels=128, use_1x1conv=True, strides=2),
            ResNetLayer(in_channels=512, out_channels=128),
            ResNetLayer(in_channels=512, out_channels=128),
            ResNetLayer(in_channels=512, out_channels=128)
        )
        self.conv4 = nn.Sequential(
            ResNetLayer(in_channels=512, out_channels=256, use_1x1conv=True, strides=2),
            ResNetLayer(in_channels=1024, out_channels=256),
            ResNetLayer(in_channels=1024, out_channels=256),
            ResNetLayer(in_channels=1024, out_channels=256),
            ResNetLayer(in_channels=1024, out_channels=256),
            ResNetLayer(in_channels=1024, out_channels=256),
        )
        self.conv5 = nn.Sequential(
            ResNetLayer(in_channels=1024, out_channels=512, use_1x1conv=True, strides=2),
            ResNetLayer(in_channels=2048, out_channels=512),
            ResNetLayer(in_channels=2048, out_channels=512),
        )
        self.pool2 = GlobalAvgPool2d()         # Add dropout before the final fully connected layers
        self.dropout2 = Dropout(p=self.dropout_rate)
        self.fc = LinearRegression(2048, num_classes, lr=self.lr, bias=self.bias)

    def forward(self, X):
        Y = self.relu(self.bn1(self.conv1(X)))
        Y = self.pool1(Y)
        Y = self.dropout1(Y)

        Y = self.conv2(Y)
        Y = self.conv3(Y)
        Y = self.conv4(Y)
        Y = self.conv5(Y)

        Y = self.pool2(Y)
        Y = Y.reshape(Y.shape[0], -1)

        Y = self.dropout2(Y)
        Y = self.fc(Y)

        return Y


    def loss(self, y_hat, y):
        return CrossEntropyError(y_hat, y)

    def configure_optimizers(self):
        return SGDFromScratch(self.parameters(), self.lr)
```

This is a huge step-up from a single Linear model in our first notebook. Back in the days of Lin's, people would handwrite (or type) this kind of code in a more, *cough cough*, archaic language like C++, ASM, Fortran or even Lisp (totally didn't rip this off from Google's AI Overview).

The advent of tooling, like PyTorch and other ML/AI libraries, shifted our focus. Instead of worrying if we are implementing each functions correctly, we can now deal with them in a high-level way. You don't have to stress about whether `CrossEntropyError` or `Conv2d` works properly, you can just drop them in high-level functions like `ResNetLayer`, which then stack into higher-level and full architecture like `ResNet50`.

These reusable abstractions are arguably one of the core reasons behind the ML/AI explosion that came before the ChatGPT.

Let us test the output of this model to verify and see if everything is working correct. We have done a lot of scaffolding and building without testing it out. In the future, we wish to be better by testing at every step.

```
model = ResNet50(num_classes=2, in_channels=3, lr=0.01, dropout_rate=0.4)

model(torch.normal(0, 1, (1, 3, 224, 224)))
```

```
tensor([[-0.2781, -0.4704]], grad_fn=<AddBackward0>)
```

3.5 Conclusion

We have condensed the Residual Learning paper in a way that we skipped other details on why certain design choices within the ResNet architecture are used. The model we used leverages the numbers of `ResNetLayers` we have to extract as much learned features for identification. Coupled with skip connections, we can immediately bypass irrelevant feature to improve both the performance and convergence of our model.

References

- [1] K. He, C. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770-778.
- [2] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*. 2021. [Online]. Available: https://d2l.ai