In [1]:
```
%%capture
! [ -e /content ] && pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

In [2]:
```
#hide
from fastai.vision.all import *
from fastbook import *

matplotlib.rc('image', cmap='Greys')
```

Manual Linear Learner Model to Classify Handwritten Digits

Extracting Data

MNIST contains images of handwritten digits. We will use these images to manually create a Le model that can detect the difference between handwritten numbers.

In [3]:
```
path = untar_data(URLs.MNIST)
Path.BASE_PATH = path
(path).ls()
```

. %[/ : < :]

Out[3]:
```
(#2) [Path('training'),Path('testing')]
```

We will download the training and validation images for each number into respective arrays, con them to a scale of to .

In [4]:
```
train_tns = []
valid_tns = []
# extract and convert to scale of 0 to 1 using tensor operations
for i in range(10):
  train_tns.append(torch.stack([tensor(Image.open(o)).float()/255 for o i
  valid_tns.append(torch.stack([tensor(Image.open(o)).float()/255 for o i
```

Now, we need to convert our tensors from a list of matrices to a list of vectors, flattening the pixe each image to a single row of pixels. We use `view` which increases the axis to as big as needed to fit all the data. Then, we will create labels for each of the images using integ encoded class labels for multi-class classification.

In [5]:
```
# extract images and flatten
train_x = torch.cat(train_tns).view(-1, 28*28)
valid_x = torch.cat(valid_tns).view(-1, 28*28)

# extract labels for each image
train_y = []
valid_y = []
for i in range(10):
  train_y.extend([i] * len(train_tns[i]))
  valid_y.extend([i] * len(valid_tns[i]))
```

```python
# make labels rank-2 tensors
train_y = tensor(train_y).unsqueeze(1)
valid_y = tensor(valid_y).unsqueeze(1)
train_x.shape, train_y.shape
```

Out[5]:

```
(torch.Size([60000, 784]), torch.Size([60000, 1]))
```

`.unsqueeze()` adds a new dimension to train_y, making it a rank- tensor, suitable for classification tasks. Because of this, it is easy to create a Dataset for PyTorch where each flatter image is associated with its label.

In [6]:

```python
dset = list(zip(train_x, train_y))
valid_dset = list(zip(valid_x, valid_y))
```

Now we are ready to begin training our Learner model.

Creating Model

Initialise Weights

We start by initialising our **weights** with random values from the normal distribution. Each weigh associated with one of the x pixel values. We give weights a shape in the nd axi , representing the probability the image belongs to each of the classes from t We also need to define bias to ensure the model is suitably flexible, otherwise input of will trainable. Together, the weights and bias define our **parameters**.

In [18]:

```python
def init_params(size, std=1.0):
  """Returns init"""
  return (torch.randn(size)*std).requires_grad_()

weights = init_params((28*28, 10))
bias = init_params(10)

params = weights, bias
weights.shape, bias.shape
```

Out[18]:

```
(torch.Size([784, 10]), torch.Size([10]))
```

Make a Prediction

Now that we have parameters and input, we can create a model that uses the parameters to cla each input.

We will use **matrix multiplication** for this task. We want to find the dot product of the weights ar pixel values, then add the bias to this. We could do this in a number of ways, including Python `for` loops - but this would be very inefficient. PyTorch is optimised to do the same task matrix multiplication (`@` operator) extremely quickly on the GPU. We add the bias to the model

In [19]:

```python
def matrix_x(x):
  """Uses the models parameters to predict the classification of an image
  return x@weights + bias
```

Calculate Loss

Now, we calculate **loss** to determine how effective our parameters are in classifying handwritten
Loss is a value that represents how well (or badly) our model is doing.

Because we are performing multi-class classification we will use **cross-entropy loss**, which qua
how well predicted probability distributions align with the true labels, penalizing incorrect predicti
based on their confidence levels.

In [20]:
```
def calc_loss(preds, y):
  """Calculates the cross-entropy loss using a tensor of logits from the
  return torch.nn.functional.cross_entropy(preds, y.squeeze())
```

Calculate Gradient

Using **stochastic gradient descent** (SGD), which is an **optimiser** function, we will update the parameters in the model in each **epoch** to improve its predictive power. An epoch refers to a sin complete pass of the entire training dataset through the model. SGD is an optimisation algorithm which the parameters are updated by finding the gradient (derivative) of each parameter and gra adjusting it so it moves towards its optimal value. PyTorch automatically employs **backpropagat** calculate the gradient of the loss with respect to each weight using the chain rule of calculus bec linear_model is defined using `torch.nn`. PyTorch can very efficiently calculate derivatives.

We calculate the loss for **mini-batches** to ensure efficiency and suitable loss predictions. We wi `DataLoader` object from PyTorch which turns a Python collection into an iterable, connecting input with its respective label. It will also shuffle the data items in each mini-batch every epoch. T **batch size** is the number of data items in a mini-batch.

In [21]:
```
# make a function to compute the models predictions, calculate the loss a
def calc_grad(xb, yb, model):
  preds = model(xb)
  loss = calc_loss(preds, yb)
  loss.backward()
```

Step the Parameters

Now we can update each parameter (called "stepping") using the gradient calculated in `calc_g` and the **learning rate** defined by `lr`. The learning rate is size of step we take when applying S update the parameters.

We will also create some code to define a metric for human consumption called accuracy. This w determine how accurately the model is correctly classifying the images and is more appropriate human evaluation than loss (which is better for the training process).

In [22]:
```
# initialise parameters and 2 DataLoader objects for training set and val
dl = DataLoader(dset, batch_size=256)
valid_dl = DataLoader(valid_dset, batch_size=256)

def train_epoch(model, lr, params):
  """Update the parameters using the gradient of loss with respect to the
```

```
  for x, y in dl:
    calc_grad(x, y, model)
    for p in params:
      p.data -= p.grad * lr
      p.grad.zero_() # reset gradient, as PyTorch accumulates by default

def batch_accuracy(x, y):
  preds = x.softmax(dim=1)
  predicted_classes = preds.argmax(dim=1)
  correct = predicted_classes == y
  return correct.float().mean()

def validate_epoch(model):
  accs = [batch_accuracy(model(xb), yb) for xb, yb in valid_dl]
  return round(torch.stack(accs).mean().item(), 4)

for i in range(20):
  train_epoch(matrix_x, 0.1, params)
  print(validate_epoch(matrix_x))
```

```
0.1763
0.2779
0.3708
0.436
0.4744
0.5022
0.5237
0.5437
0.5571
0.5701
0.5803
0.5882
0.5981
0.605
0.6114
0.6168
0.6223
0.628
0.6332
0.6374
```

This is now a working linear learner model working at about % accuracy. It is not yet a neu network as we have not introduced non-linearity - this is expected to drastically improve its predi power.

Activation Function

For the model to improve, we need to add more **layers** and non-linearity through **activation fun** Linear classifiers are constrained in terms of their predictive power - to make it more complex to more tasks we need to make it a **neural network** (NN). A NN is a computational model inspired human brain, consisting of layers of interconnected nodes (neurons) that process and transform data through weighted connections, enabling it to learn complex patterns and make predictions c classifications.

We will do this by making our model layers.

. Layer will be a **linear layer**.

. Layer will be an **activation layer**.

. Layer will be a **linear layer**.

A **linear layer** also known as a **fully connected layer** linearly transforms the input by applying tl parameters. This is what our linear model above does through the equation `y = WX + b` whe

- `y` is the transformed data.
- `W` is the weights matrix.
- `b` is the bias vector.
- `X` is the input features.

An **activation layer** introduces non-linearity in the model, enabling it to learn and model comple: patterns. For example, the **Rectified Lienar Unit** (ReLU) function is a common activation functic (makes negative values).

In [29]:
```
simple_net = nn.Sequential (
    nn.Linear(28*28, 30), # Layer 1
    nn.ReLU(), # Layer 2
    nn.Linear(30, 10) # Layer 3
)
```

In the first layer, we have weights for each pixel in the MNIST images. We used as a somewhat arbtirary hyperparameter (the number of neurons) as the output size. We would refine this parameter through experimentation. This value allows the model to learn more comple patterns.

In the second layer, we use the ReLU activation function to add nonlinearity.

In the third layer, we have input, representing the **activations** from layer . Activations are the numbers that are calculated and returned by each linear or activation (non-lir layer. The output size of corresponds to the number of classes in the MNIST dataset ( ). This layer will output a vector of values, representing the network's "confidence" (**lo** each class.

We will be using the fastai model to make the NN. As such, we need to define a `DataLoaders` with our training and validation sets.

In [33]:
```
dls = DataLoaders(dl, valid_dl)
```

In [35]:
```
learn = Learner(dls, simple_net, opt_func=SGD, loss_func=calc_loss, metri
learn.fit(20)
```

epoch	train_loss	valid_loss	batch_accuracy	time
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:
	.	.	.	:

By using a NN instead of a linear learner model, we have been able to improve the accuracy to a %. If we wanted to further improve our model, we could utilise pre-trained models such as resnet . However, this process has demonstrated the predictive power of the NN.

While the model achieved reasonable accuracy, there is a risk of **overfitting**, especially as the m becomes more complex.Overfitting occurs when a model learns to perform exceptionally well on training data but fails to generalize to new, unseen data, often capturing noise or irrelevant patte instead of the underlying trends.

Conclusion

This is why deep learning seems magical:

. The neural network can solve any problem to any level of accuracy given the correct set of parameters.

. There is a way to find the best set of parameters for any function called stochastic gradient