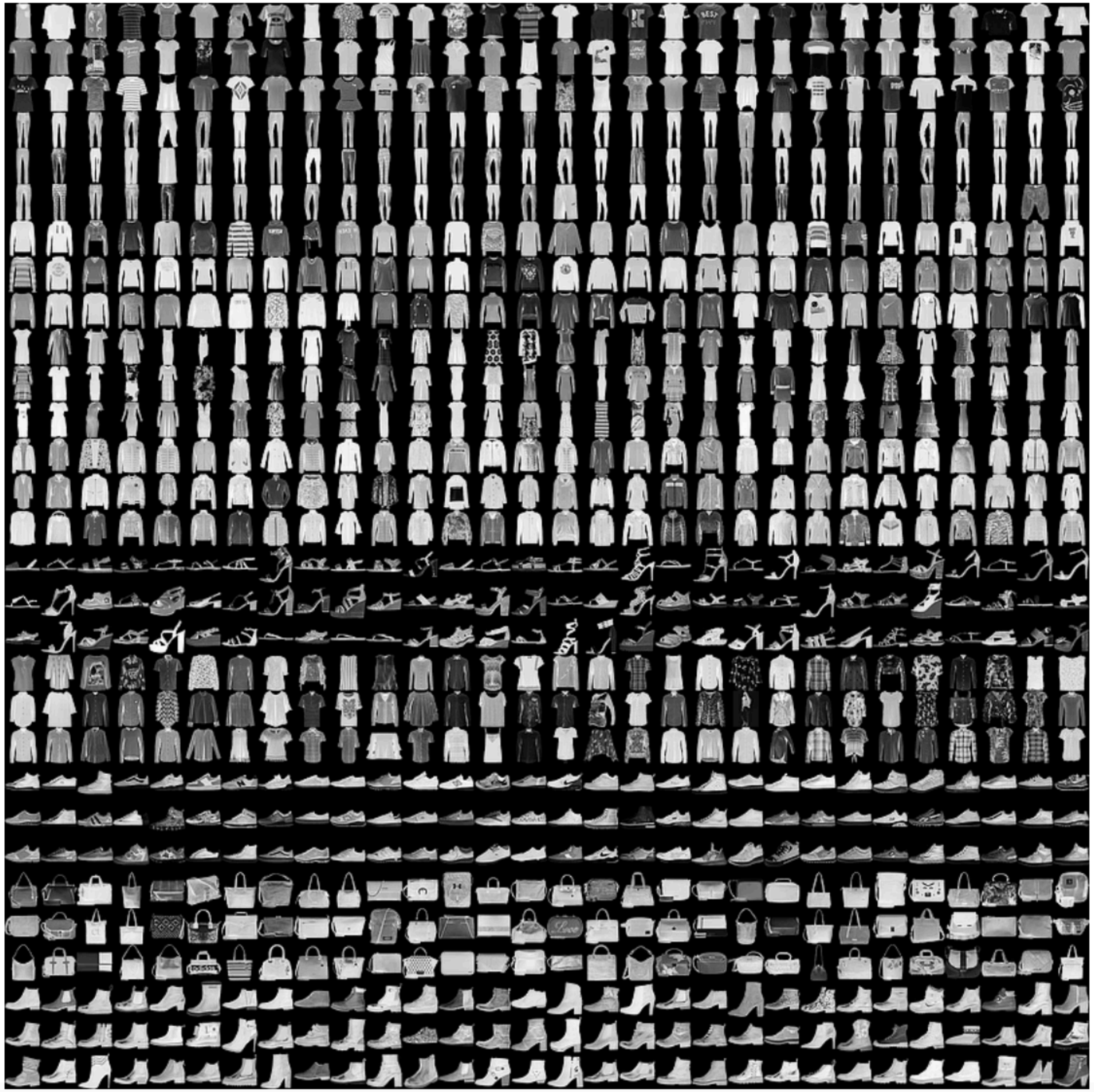
Concise Implementation of Softmax Regression

- Goal: use high-level APIs of PyTorch for implementing Softmax Regression for classification.

```
In [65]: batch_size = 256
         train_iter, test_iter = mu.load_data_fashion_mnist(batch_size)
         # type(train_iter)
```

```
In [66]: X, y = next(iter(train_iter)) # first batch
         print(X.size())
         print(y)
```

```
torch.Size([256, 1, 28, 28])
tensor([7, 8, 2, 2, 1, 2, 6, 8, 5, 5, 7, 3, 0, 8, 7, 5, 9, 7, 9, 5, 8, 5, 9,
3,
        5, 0, 6, 1, 7, 4, 3, 2, 6, 5, 1, 5, 5, 1, 4, 2, 6, 2, 9, 9, 8, 5, 6,
8,
        2, 8, 6, 8, 6, 7, 8, 2, 2, 7, 9, 8, 5, 3, 8, 8, 6, 5, 8, 0, 4, 9, 2,
6,
        9, 7, 2, 5, 1, 8, 5, 6, 6, 7, 4, 5, 0, 8, 2, 5, 5, 5, 0, 4, 1, 8, 4,
9,
        1, 8, 1, 1, 6, 4, 4, 7, 8, 7, 8, 7, 8, 7, 0, 7, 1, 9, 7, 8, 6, 1, 9,
6,
        3, 8, 6, 3, 7, 7, 3, 0, 6, 9, 1, 4, 2, 2, 8, 2, 6, 1, 0, 1, 2, 2, 0,
0,
        8, 2, 7, 5, 1, 0, 5, 5, 9, 4, 2, 5, 2, 1, 9, 5, 3, 9, 5, 3, 8, 2, 1,
2,
        8, 3, 6, 0, 6, 6, 2, 9, 6, 6, 0, 5, 1, 6, 6, 3, 9, 7, 4, 8, 8, 6, 7,
0,
        2, 4, 6, 3, 4, 7, 5, 3, 8, 1, 0, 4, 0, 5, 7, 5, 4, 2, 7, 9, 3, 7, 0,
8,
        6, 7, 6, 2, 9, 8, 7, 2, 6, 0, 1, 9, 1, 3, 7, 4, 6, 0, 4, 8, 4, 8, 4,
1,
        3, 0, 1, 1, 3, 2, 2, 9, 0, 2, 6, 9, 6, 8, 3, 9])
```



Defining the Model and Initialization

- Each example is represented by a fixed-length vector: we flatten each 28×28 image, treating it as vector of length 784.
- Because our dataset has 10 classes, our network will have an output dimension of 10.
- So, our weights `W` will be a 784×10 matrix and the biases `b` will constitute a 10×1 row vector.
- We initialize `W` using a Gaussian distribution and `b` with 0.
- Softmax regression can be implemented as a Fully-Connected (i.e Linear) layer.

In [91]:

```python
class Net(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(Net, self).__init__()
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs
        self.Linear1 = nn.Linear(num_inputs, num_outputs)
        torch.nn.init.normal_(self.Linear1.weight, std=0.01) #init the weights
        torch.nn.init.zeros_(self.Linear1.bias) #init the bias

    def forward(self, x):
        x = x.view(-1, self.num_inputs)
        out = self.Linear1(x)
        return out

num_inputs, num_outputs = 784, 10
net = Net(num_inputs, num_outputs)
print(net)
```

```
Net(
  (Linear1): Linear(in_features=784, out_features=10, bias=True)
)
```

Alternative Initialization

- This is useful if you have multiple layers of the same type and you want them to be initialized in the same way.

In [93]:

```
def init_weights(m):
    if isinstance(m, nn.Linear): # by checking type we can init different layers in different ways
        torch.nn.init.normal_(m.weight, std=0.01)
        torch.nn.init.zeros_(m.bias)

net.apply(init_weights);
# print(net)
```

Loss Function

- Use PyTorch's implementation of Softmax-Cross Entropy loss to avoid numerical instabilities.
  - The input to loss function are the logits logits $\mathbf{o}$ (and not softmax outputs)

In [94]:
```
loss = nn.CrossEntropyLoss()
```

Optimization Algorithm

- Minibatch SGD with a learning rate of 0.1 as the optimization algorithm.

In [95]:
```
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

Evaluation

In [96]:
```
def accuracy(y_hat, y):  #y_hat is a matrix; 2nd dimension stores prediction sco
res for each class.
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1) # Predicted class is the index of max score
    cmp = (y_hat.type(y.dtype) == y)  # because`==` is sensitive to data types
    return float(torch.sum(cmp)) # Taking the sum yields the number of correct p
redictions.

# Example: only 1 sample is correctly classified.
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
accuracy(y_hat, y) / len(y)
```

Out[96]:
```
0.5
```

In [97]:
```
class Accumulator:
    """For accumulating sums over `n` variables."""
    def __init__(self, n):
        self.data = [0.0] * n # [0, 0, ..., 0]
    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]
    def reset(self):
        self.data = [0.0] * len(self.data)
    def __getitem__(self, idx):
        return self.data[idx]
```

In [98]:
```
def evaluate_accuracy(net, data_iter):
    """Compute the accuracy for a model on a dataset."""
    metric = Accumulator(2)  # No. of correct predictions, no. of predictions
    for _, (X, y) in enumerate(data_iter):
        metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

- The accuracy of the model prior to training should be close to random guessing, i.e., 0.1 for 10 classes.

In [99]:
```
evaluate_accuracy(net, test_iter)
```

Out[99]:
```
0.1459
```

Training

- The training loop for softmax regression looks strikingly familiar with that of linear regression.
- Here we refactor the implementation to make it reusable.
    - First, we define a function to train for one epoch.

In [100]:

```
def train_epoch_ch3(net, train_iter, loss, optimizer):
    """The training function for one epoch."""
    # Set the model to training mode
    if isinstance(net, torch.nn.Module):
        net.train()
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        y_hat = net(X)
        l = loss(y_hat, y)
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
        metric.add(float(l) * len(y), accuracy(y_hat, y), y.size().numel())
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]
```

Training

- The following class will be used to plot training and validation accuracy as well as loss evolution over training loop.

In [101]:
```
class Animator:  #@save
    """For plotting data in animation."""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        mu.use_svg_display()
        self.fig, self.axes = mu.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda function to capture arguments
        self.config_axes = lambda: mu.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
```

Training

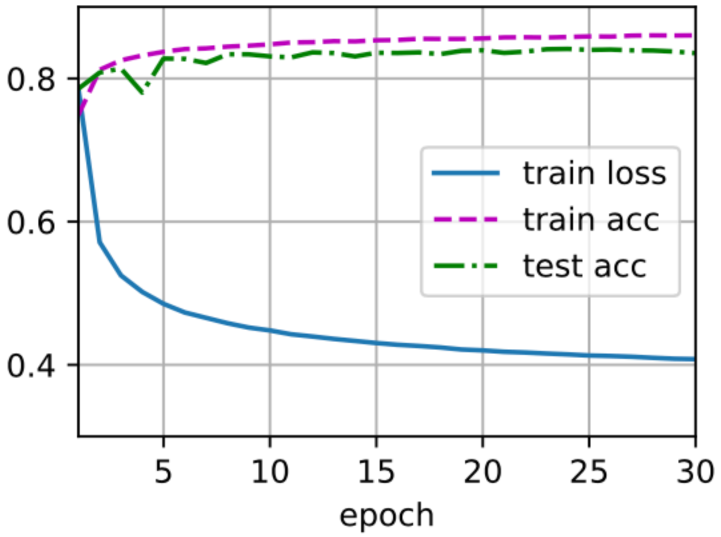
- The following function trains the model (`net`) on a training set (`train_iter`) for `num_epochs`.
- At the end of each epoch, the model is evaluated on a testing set (`test_iter`).
- `Animator` for visualizing the training progress.

In [102]:

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, optimizer):  #@save
    """Train a model."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, optimizer)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

In [103]:

```
num_epochs = 30
train_ch3(net, train_iter, test_iter, loss, num_epochs, optimizer)
```



Summary

- Using PyTorch's high-level APIs, we can implement softmax regression much more concisely.
- From a computational perspective, implementing softmax regression has intricacies.
- Note that in many cases, PyTorch takes additional precautions beyond these most well-known tricks to ensure numerical stability