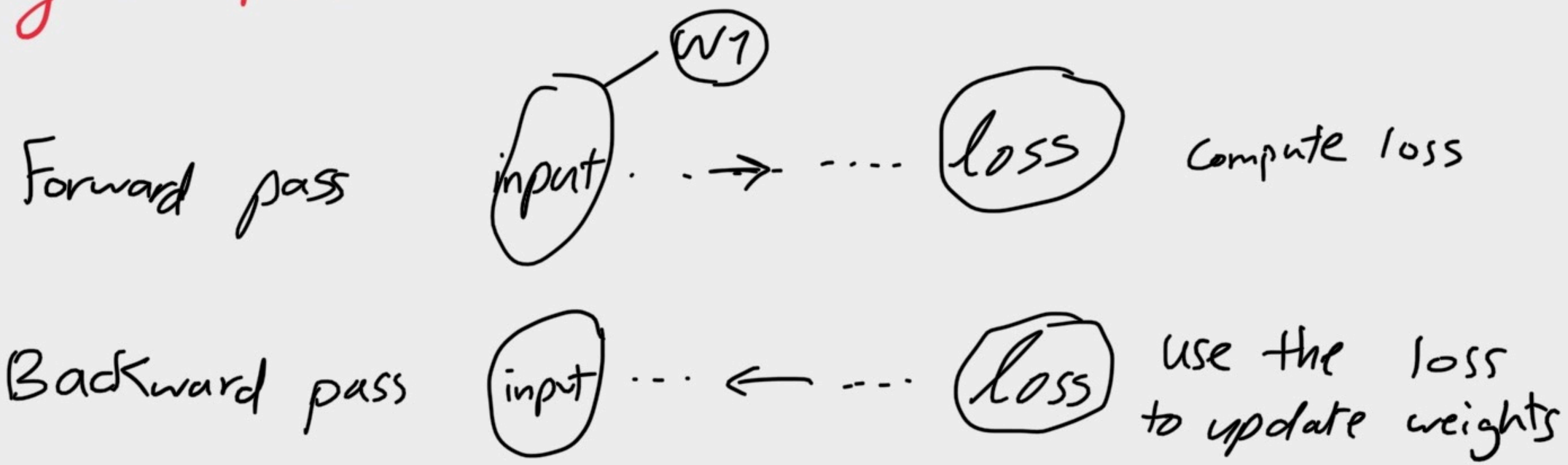
Training Neural Networks

loss function ⟶ measure prediction error

$$l = \frac{1}{2n} \sum_{i}^{n} (y_i - \hat{y}_i)^2$$

- n: number of training examples
- y_i: true labels
- $\hat{y}_i$: predicted labels



Loss in Pytorch:

nn.CrossEntropyLoss → nn.LogSoftmax()
nn.CrossEntropyLoss → nn.NLLLoss()

input: scores (not probabilities)

In summary

1. Build a feed forward network:

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU,
                      nn.Linear(128, 64),
                      nn.ReLU,
                      nn.Linear(64, 10))
```

ReLU is almost always used for hidden layers

2. Define the loss:

```
criterion = nn.CrossEntropyLoss()
```

and set this to nn.NLLLoss()

we can add a `nn.LogSoftmax(dim=1)`

3. Get the data:

```
images, labels = next(iter(trainloader))
```

4. flatten images:

```
images = images.view(images.shape[0], -1)
```

5. forward pass — get logits

```
logits = model(images)
```

6. Calculate the loss:

loss = criterion(logits, labels)

Autograd (Backpropagation)

track → `x = torch.zeros(1, requires_grad=True)`

- with `torch.no_grad()`:
  - turns off autograd

we can get the grad_fn with `x.grad_fn`

finally, we do `z.backward()`

Loss and Autograd together:

```
Code as before
:
;
loss = .....
```

`loss.backward()` ⟶ compute gradients

Using gradients:

before forward pass ↰ `optimizer.zero_grad()`

we use optimizers : ↳ `optimizer.step()` after getting the loss

Summary of Training Steps:

1. forward pass
2. compute loss
3. `loss.backward()` to compute gradients
4. take a step with optimizer to update weights

Full training code:

```
criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        images = images.view(images.shape[0], -1)
    * optimizer.zero_grad()

    Forward pass

    * loss.backward()
    * optimizer.step()

        running_loss += loss.item()
    else:
        training_loss = running_loss/len(trainloader)
```

Inference and Validation

we had this code for testing the prediction:

Validation step

```
model = Classifer()
images, labels = next(iter(testloader))
ps = torch.exp(model(images))
```

→ (64, 10)

64 → #images

10 → #classes

```
top_p, top_class = ps.topk(1, dim=1)
```

k → k highest value

```
equals = top_class == labels.view(*top_class.shape)

accuracy = torch.mean(equals.type(torch.FloatTensor))
```

because equals is a bit tensor and we have to convert it.

Implementing Validation Pass:

```
for e in epochs:
    for image, label in trainloader:
        as before

    else:
        with torch.no_grad():
            above code
```

the validation pass