4.1 Layer initialization and transfer learning

```
# initialize layer weights

import torch.nn as nn
layer = nn.Linear(64, 128)
nn.init.uniform_(layer.weight)
# distribution is now from 0 to 1
```

```
import torch

layer = nn.Linear(64, 128)
torch.save(layer, 'layer.pth')
new_layer = torch.load('layer.pth')
```

Fine-tuning is a type of transfer learning, we can import weights from previous layers and then use a smaller learning rate. We then train part of the network (so freeze early layers)

```
import torch.nn as nn
model = nn.Sequential(nn.Linear(64, 128), nn.Linear(128, 256))

for name, param in model.named_parameters():
    if name == '0.weight':
        param.requires_grad = False
```

4.2 Evaluating model performance

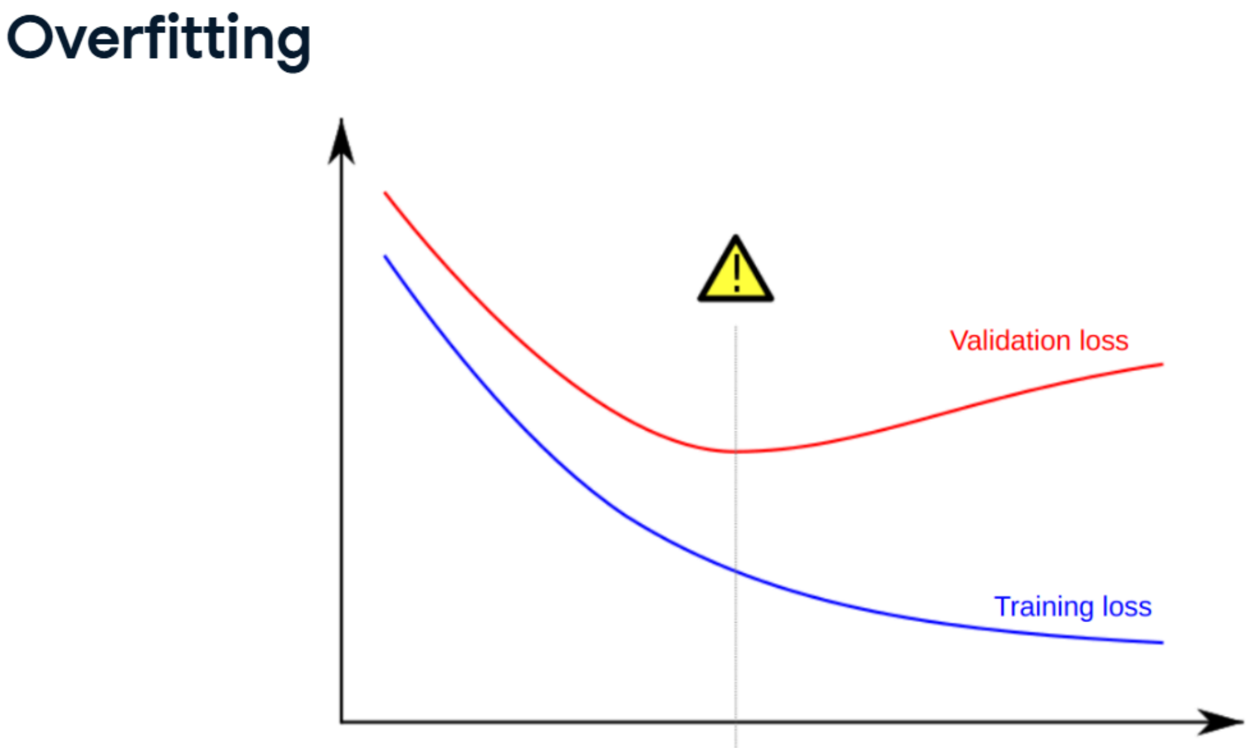
Training adjusts model parameters
Validation tunes hyperparameters
Test evaluates final model performance

Compute the mean training loss at the end of each epoch

```
training_loss = 0.0

for inputs, labels in trainloader:
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step() # update weights
    optimizer.zero_grad() # reset weights
    training_loss += loss.iten()

epoch_less = training_loss / len(trainloader) # calculate mean loss
```

```
validation_loss = 0.0
model.eval() # put model in evaluation mode

with torch.no_grad(): # disable gradeints for efficiency
    for inputs, lables in validationloadeR:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        validation += loss.item()

epoch_loss = validation_loss / len(validationloader) # calculate mean loss
model.train() # switch back to training mode
```

This is an overfitting graph



Loss does not reflect accuracy in predictions

```
import torchmetrics

metric = torchmetrics.Accuracy(task="multiclass", num_classes=3)

for features, labels in dataloader:
    outputs = model(features) # forward pass
    # compute batch accuracy, keeping argmax for one-hot labels
    metric.update(outputs, labels.argmax(dim=-1))

accuracy = metric.compute() # compute accuracy over the whole epoch
metric.reset() # clear state before next epoch
```

4.3 Fighting overfitting

Overfitting: the model does not generalize to unseen data

Strategies to fighting overfitting:

- **reducing model size**
- **adding dropout layer** - regularization technique that randomly deactivates neurons at random

```
model = nn.Sequential(nn.Linear(8,4)
                      nn.ReLU(),
                      nn.Dropout(p=0.5)) # probability = 50%
features = torch.randn((1,8))
```

- **use weight decay to force parameters to remain small**

```
optimizer = optim.SGD(model.parameters(), lr=0.001, weight_decay=0.0001)
# encourages smaller weights by adding a penatly during optimization
# helps reduce overfitting, keeping weights smaller and improving
generalization
```

- **obtain new data or augmenting data**

New data is expensive. We can augment data by rotating or scaling the data to generate "new" data points

4.4 Improving model performance

Recipe for tackling any deep learning problem

1. model to overfit the training set (ensure the problem is solvable)
2. set a performance baseline
3. reduce overfitting to increase performance on the validation set
4. achieve the best possible performance by tuning hyperparameters

```
# 1
features, labels = next(iter(dataloader))
for i in range(1000):
    outputs = model(features)
    loss = criterion(outputs, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

2. Experiment with:
    1. dropout
    2. data augmentation
    3. weight decay
    4. reducing model capacity

We should balance reducing overfitting strategies and regularization

3. Fine-tune hyperparameters

```
# grid search
for factor in range(2,6):
    lr = 10 ** -factor
```

```
# random search (optimal)
factor = np.random.uniform(2,6)
lr = 10 ** -factor
```