00:00 - Tweaking first and last layers

02:47 - What are the benefits of using larger models

Understands smaller features

05:58 - Understanding GPU memory usage

Larger models can easily deprive GPUs of their memories. Then restart your notebook and by following the below procedure, we can avoid such errors. Just to check the memory usage, we can take the category that is the smallest in the dataset and only use them on different architectures and compare the results of GPU memory consumption. Because, GPU memory doesn't depend on the overall training dataset size and thus how long the training lasts for. It depends on image size and batch size, which are the same as taking the least possible images we can use for training the model architectures.

08:04 - What is GradientAccumulation?

```
def train(arch, size, item=Resize(480, method='squish'), accum=1, finetune=True, epochs=12):
    dls = ImageDataLoaders.from_folder(trn_path, valid_pct=0.2, item_tfms=item,
        batch_tfms=aug_transforms(size=size, min_scale=0.75), bs=64//accum)
    cbs = GradientAccumulation(64) if accum else []
    learn = vision_learner(dls, arch, metrics=error_rate, cbs=cbs).to_fp16()
    if finetune:
        learn.fine_tune(epochs, 0.01)
```

```
for x,y in dl:
    calc_loss(coeffs, x, y).backward()
    coeffs.data.sub_(coeffs.grad * lr)
    coeffs.grad.zero_()
```

Here's the same thing, but with gradient accumulation added (assuming a target effective batch size of 64):

```
count = 0            # track count of items seen since last weight update
for x,y in dl:
    count += len(x)  # update count based on this minibatch size
    calc_loss(coeffs, x, y).backward()
    if count>64:     # count is greater than accumulation target, so do weight update
        coeffs.data.sub_(coeffs.grad * lr)
        coeffs.grad.zero_()
        count=0      # reset count
```

First, we divide the batch size by the accumulation factor. So, this decreases the memory consumption by the GPU. However, this has its own implications of being computationally ineffective to vary the gradients on very small batches.

So, instead of updating and zeroing out the gradients in every step, if we just loop through calculating the loss and loss.backward() in pytorch, we can see that the gradients of the corresponding two batches add up. This would be the same as having 64 images at once and then updating the parameters and zeroing out the gradients, however using less GPU memory.

Batch normalization, which we discuss at a later point, causes this to produce different results. However, not bad results. And many models don't use batch normalization and all those models produce indifferent results upon using Gradient accumulation.

Standard approach to pick a batch size -> use largest as you can . Jeremy uses smaller than necessary.

We also need to change the learning rate a little, to adjust for the change in batch sizes. Smaller batch sizes would need smaller learning rates.

21:46 - garbage collection and empty cache:

It is a good practice to do this after testing our models as this frees up the GPU after usage. Prevents GPU memory fragmentation.

22:55 - Ensembling

```
    dls = ImageDataLoaders.from_folder(trn_path, valid_pct=0.2, item_t
fms=item,
        batch_tfms=aug_transforms(size=size, min_scale=0.75), bs=64//a
ccum)
```

Data loaders don't have a random seed. So, while using different architecture testing, we are not using the same training and validation sets. Then we are using Ensembling -> Bagging.

Train and test different models and average the results out.

We can also weigh the results of a certain model in the Ensemble if a certain model works well.

Ensembling Vs K-fold cross validation:K-fold cross validation is to have non-overlapping validation sets through different runs. In theory K-fold cross validation works well. But, in Jeremy's experience, using more architectures and ensembling is much more fruitful.

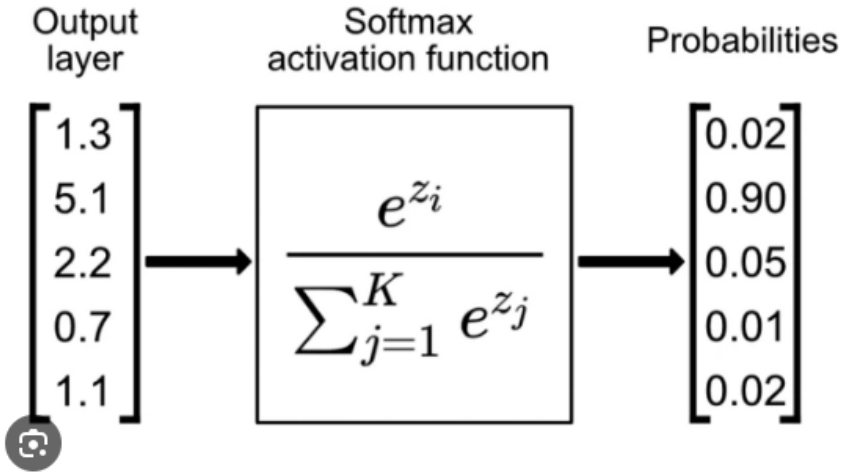
37:51 - Multi-target models

What if we need to predict the paddy disease as well as the variety of the rice?
If there are 10 diseases and 10 varieties -> we just need to create 20 activations in the final softmax layer and for the first 10 activations, we need to specify the loss function as cross entropy loss between predictions and diseases and the loss function corresponding to next 10 activations is set as the cross entropy loss between predictions and varieties. So, the first 10 predictions correspond to the probabilities regarding the disease classification and the next 10 probabilities correspond to the probabilities of the variety classification. The overall loss would be the sum of both these losses.

41:24 - What does `F.cross_entropy` do

45:43 - When do you use softmax and when not to?

When using softmax, it is imperative that the object of interest belongs to one category or the other as the total sum of all the probabilities of the classes is one.



So, sometimes, when that is not what we want, we can also use a different approach where the sum of outputs(in this case the probabilities) is less than one(Thereby leaving the possibility of the object's classification into none of the classes) or even more than one, when there are multiple objects you are detecting and hence, having multiple probability values.

46:15 - Cross_entropy loss

49:53 - How to calculate binary-cross-entropy

52:19 - Two versions of cross-entropy in pytorch

58:00 - Multi-target model Vs single-target-model

When we train this model that is predicting both the disease and variety gets better at detecting disease as well - Better performance in disease detection than a model that only detects disease. The reason is shared features may give it a better understanding.

So, **sometimes**, a multi-target model gives a better output than a single-target model.

1:02:00 - Collaborative filtering deep dive

Consider a website such as NETFLIX, where different users watch different movies. Now, what can be the recommended movies list for this specific user. We get this by understanding the themes of the movies that this specific user likes and then see what movies did other users with similar taste watch that our user1 hasn't watched yet.

1:08:55 - What are latent factors?

```
last_skywalker = np.array([0.98,0.9,-0.9])
```

```
user1 = np.array([0.9,0.8,-0.6])
```

Let us assume that the in the array corresponding to the last_skywalker movie, the first element refers to the genre, science fiction, the second element corresponds to the film belonging to action genre, and the third element being an indicator of how old the movie is.So, here we can see that the movie last_skywalker has science fiction and action themes in it, and a fairly new movie.

The numbers corresponding to user1, contains how strong his sentiments are with respect to each feature in the array. So, these values say that the user1 likes science fiction and action movies and don't have a strong preference to old classics.

```
(user1*last_skywalker).sum()

2.1420000000000003
```

This high number suggests that the movie last_skywalker sits well with his interests and sentiments.

On the other hand, the movie Casablanca can be represented as:

```
casablanca = np.array([-0.99,-0.3,0.8])
```

It is not of a particularly science fiction and action genre and fairly new.

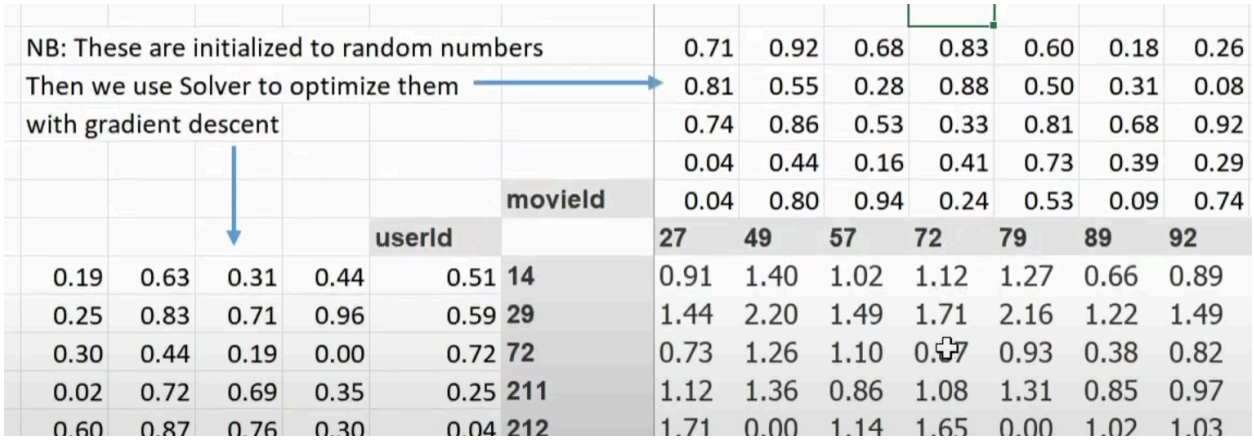
```
(user1*casablanca).sum()

-1.611
```

This gives us a negative value as it does not sit well with the interests of user1.

So, given the proclivities of the user and the corresponding themes of the movies, we can get a quantifiable number that shows how well the user is interested in the movie. But, how can we get these features that correspond to themes and user proclivities? These are called **latent factors.** We don't know them in advance.

	movieId														
	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
userId 14	3.0	5.0	1.0	3.0	4.0	4.0	5.0	2.0	5.0	5.0	4.0	5.0	5.0	2.0	5.0
29	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	4.0	5.0	5.0	3.0	4.0	5.0
72	4.0	5.0	5.0	4.0	5.0	3.0	4.5	5.0	4.5	5.0	5.0	5.0	4.5	5.0	4.0
211	5.0	4.0	4.0	3.0	5.0	3.0	4.0	4.5	4.0		3.0	3.0	5.0	3.0	
212	2.5		2.0	5.0		4.0	2.5		5.0	5.0	3.0	3.0	4.0	3.0	2.0
293	3.0		4.0	4.0	4.0	3.0		3.0	4.0	4.0	4.5	4.0	4.5	4.0	
310	3.0	3.0	5.0	4.5	5.0	4.5	2.0	4.5	4.0	3.0	4.5	4.5	4.0	3.0	4.0

This is the list of users and the movies they have watched. The values in the table correspond to the ratings of the respective users for the movie. Empty values -> user not yet watched the movie.



The five values corresponding to each movie are the latent features that are considered to be important to understand the themes and subject of the movie and the five values corresponding to each user corresponding to user's proclivities for each of these features respectively.

In other words, the five values corresponding to the movie are the features and the five values corresponding to the user are the weights. The dot product or matrix multiplication gives us the value -> prediction value. So, we can initiate the features and the user proclivity weights with random variables and then compare the cross product result with the actual rating values given by the users. Optimizing this loss function, will give us the best values that reflect the original data.

So, using these values, after optimization, we get different movie themes and how users feel about them. Now, using these weights and features, we can also calculate the scores or ratings of the empty values and suggest the movies to the respective users if the rating predictions are high. Effectively, COLLABORATIVE FILTERING is a matrix completion exercise, where, it is observed that users with similar interests liked these movies, so you might like them too.

1:18:37 - What is embedding

original data				user embedding						movie embedding					
userId	**movieId**	**rating**	**user idx**	*1*	*2*	*3*	*4*	*5*	**movie idx**	*1*	*2*	*3*	*4*	*5*
14	*417*	**2**	1	0.19	0.63	0.31	0.44	0.51	14	0.75	0.47	0.05	0.91	0.59
29	*417*	**4**	2	0.25	0.83	0.71	0.96	0.59	14	0.75	0.47	0.05	0.91	0.59
72	*417*	**5**	3	0.30	0.44	0.19	0.00	0.72	14	0.75	0.47	0.05	0.91	0.59
211	*417*	**3**	4	0.02	0.72	0.69	0.35	0.25	14	0.75	0.47	0.05	0.91	0.59
212	*417*	**3**	5	0.60	0.87	0.76	0.30	0.04	14	0.75	0.47	0.05	0.91	0.59

These corresponding values of the user proclivities and themes are called embeddings, user embeddings and movie embeddings respectively.

EMBEDDING -> looking something up in an array

jargon: Embedding: Multiplying by a one-hot-encoded matrix, using the computational shortcut that it can be implemented by simply indexing directly. This is quite a fancy word for a very simple concept. The thing that you multiply the one-hot-encoded matrix by (or, using the computational shortcut, index into directly) is called the *embedding matrix*.

1:22:18 - How do you choose the number of latent factors

Different applications would need a different number of latent factors. Jeremy, proposed a function that seems to work well to find the number of latent factors with respect to the case at hand correspondingly.

1:27:13 - How to build a collaborative filtering model from scratch

```
class DotProduct(Module):
    def __init__(self, n_users, n_movies, n_factors, y_range=(0,5.5)):
        self.user_factors = Embedding(n_users, n_factors)
        self.movie_factors = Embedding(n_movies, n_factors)
        self.y_range = y_range

    def forward(self, x):
        users = self.user_factors(x[:,0])
        movies = self.movie_factors(x[:,1])
        return sigmoid_range((users * movies).sum(dim=1), *self.y_range)
```

1:29:57 - How to understand the `forward` function

1:32:47 - Adding a bias term

Low user bias ->a user gives less ratings typically than an average user due to his preferences

High user bias -> A user gives high ratings typically than an average user

High movie bias -> Movie seems to be liked typically by majority of users than expected

Low movie bias -> Movie seems to be less liked typically by majority of users than expected

Let's create a simplified example with user and movie embeddings, including bias terms. We will use a few latent factors and demonstrate how the embeddings and biases work together to predict a user's rating for a movie.

Scenario:

- **Number of Users:** 2
- **Number of Movies:** 2
- **Number of Latent Factors:** 2 (for simplicity)
- **Bias Terms:** Included for both users and movies

Step 1: Define Embeddings and Biases

User Embeddings

- **User 1 Embedding:** [0.5, 0.2]
- **User 2 Embedding:** [0.3, 0.8]
- **User 1 Bias:** -0.2 (tends to rate lower than average)
- **User 2 Bias:** +0.3 (tends to rate higher than average)

Movie Embeddings

- **Movie A Embedding:** [0.6, 0.4]
- **Movie B Embedding:** [0.1, 0.9]
- **Movie A Bias:** +0.5 (generally well-liked)
- **Movie B Bias:** -0.3 (generally less liked)

Step 2: Predict Ratings Using Matrix Multiplication and Biases

Rating Prediction Formula:

$$\text{Predicted Rating} = \text{User Embedding} \times \text{Movie Embedding} + \text{User Bias} + \text{Movie Bias}$$

Calculate for Each User-Movie Pair:

1. **User 1 - Movie A:**
   - User 1 Embedding: [0.5, 0.2]
   - Movie A Embedding: [0.6, 0.4]
   - Matrix Multiply: $(0.5 \times 0.6) + (0.2 \times 0.4) = 0.3 + 0.08 = 0.38$
   - Add Biases: $0.38 + (-0.2) + 0.5 = 0.38 - 0.2 + 0.5 = 0.68$
   - **Predicted Rating:** 0.68
2. **User 1 - Movie B:**
   - User 1 Embedding: [0.5, 0.2]
   - Movie B Embedding: [0.1, 0.9]
   - Matrix Multiply: $(0.5 \times 0.1) + (0.2 \times 0.9) = 0.05 + 0.18 = 0.23$
   - Add Biases: $0.23 + (-0.2) + (-0.3) = 0.23 - 0.2 - 0.3 = -0.27$
   - **Predicted Rating:** -0.27
3. **User 2 - Movie A:**
   - User 2 Embedding: [0.3, 0.8]
   - Movie A Embedding: [0.6, 0.4]
   - Matrix Multiply: $(0.3 \times 0.6) + (0.8 \times 0.4) = 0.18 + 0.32 = 0.5$
   - Add Biases: $0.5 + 0.3 + 0.5 = 0.5 + 0.3 + 0.5 = 1.3$
   - **Predicted Rating:** 1.3
4. **User 2 - Movie B:**
   - User 2 Embedding: [0.3, 0.8]
   - Movie B Embedding: [0.1, 0.9]
   - Matrix Multiply: $(0.3 \times 0.1) + (0.8 \times 0.9) = 0.03 + 0.72 = 0.75$
   - Add Biases: $0.75 + 0.3 - 0.3 = 0.75 + 0.3 - 0.3 = 0.75$
   - **Predicted Rating:** 0.75

Step 3: Analyze the Results

Predicted Ratings:

- **User 1 - Movie A:** 0.68
- **User 1 - Movie B:** -0.27
- **User 2 - Movie A:** 1.3
- **User 2 - Movie B:** 0.75

Interpretation:

1. **User 1 - Movie A (0.68):**
    - User 1 generally rates lower (bias = -0.2), but Movie A is generally liked (bias = +0.5). The interaction of User 1's preference (through embeddings) with Movie A's characteristics results in a slightly positive predicted rating of 0.68.
2. **User 1 - Movie B (-0.27):**
    - Even though User 1 rates low, Movie B is less liked in general (bias = -0.3). The interaction of User 1's preferences with Movie B's characteristics leads to a negative predicted rating, indicating a strong dislike.
3. **User 2 - Movie A (1.3):**
    - User 2 tends to rate higher (bias = +0.3), and Movie A is generally liked (bias = +0.5). The embeddings' interaction with the positive biases leads to a high predicted rating of 1.3, indicating strong approval.
4. **User 2 - Movie B (0.75):**
    - Even though Movie B has a negative bias (-0.3), User 2's higher rating tendency (+0.3) and the interaction of their preferences with the movie's characteristics lead to a moderately positive predicted rating.

Conclusion:

- **Bias Terms:** These account for general tendencies (user or movie) and adjust the predictions accordingly.
  - User 1's lower ratings are mitigated by Movie A's positive bias, but not enough to favor Movie B.
  - User 2's higher ratings enhance Movie A's already positive reception and moderate the less liked Movie B.
- **Embeddings:** Capture specific preferences and characteristics, influencing how users and movies interact. The combination of user and movie embeddings with bias terms helps produce a more personalized and accurate prediction.

This example demonstrates how embeddings and biases work together to refine predictions, leading to a better understanding of each user's true sentiment toward a movie.

1:34:29 - Model interpretation

We have observed that the training loss is reducing significantly, but, the validation loss is still high. This is an example of overfitting.

1:39:06 - What is weight decay and How does it help

L2 Regularization or weight decay
Adding parameters^2 to the loss function

```
parameters.grad += wd * 2 * parameters
```

Loss function update using gradients

1:43:47 - What is regularization