2021 Fast.ai Community Course

QUEENSLAND HUB

QUEENSLAND AI

Lesson 4 – Putting it together: the MNIST dataset

Mentor Introduction

- How to reach us?
  - @thanhtrv
  - @huyennguyen
- How can we help you?
  - Pytorch, graph neural networks
  - Computer vision

Key concepts

- Cross entropy loss function
- Backpropagation
- Stochastic Gradient Descent
- Minibatch
- Learning rate finder
- Fine tuning

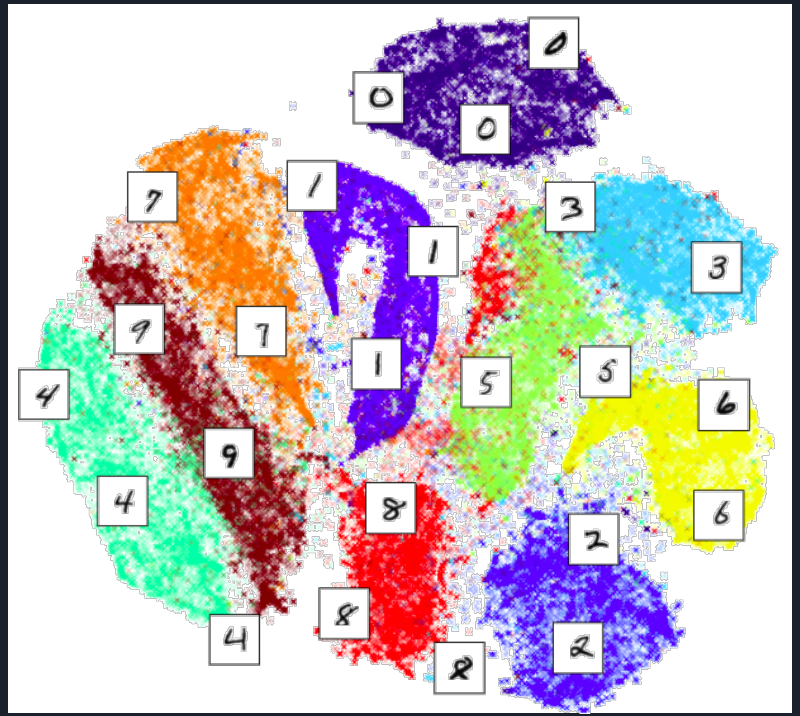
Fastbook notebooks for this week:

- 04_mnist_basics.ipynb
- 05_pet_breeds.ipynb

MNIST baseline model

- The baseline model measures the distance between a new image to the average “3” and “7” images
- What can go wrong with this model?
- Is it a useful baseline model for cats vs dogs?

Dimensionality reduction of the MNIST dataset



- “3” and ”7” are quite far apart, hence easy to discriminate
- Can this dimensionality reduction work on ImageNet?

tSNE on Imagenet

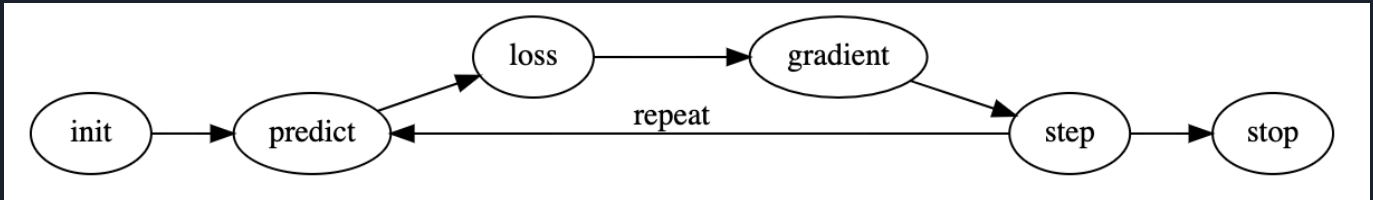
- Dimensionality reduction cannot apply directly on images (number of dimensions too high compared to the number of data points) >> curse of dimensionality
- Lower dimension representations (e.g: 4096-dimension fc7 CNN features) can be used as input into t-SNE to compute a 2-d embedding that respects the high-dimensional (L2) distances.
- Can see cluster of similar images close together
- https://cs.stanford.edu/people/karpathy/cnnembed/cnn_embed_4k.jpg

Steps in training a neural network

1. Initialize the weights.
2. For each image, use these weights to make a prediction
3. Based on these predictions, calculate how good the model is (its loss).
4. Calculate the gradient, which measures for each weight, how changing that weight would change the loss
5. Change all the weights based on that calculation.
6. Go back to the step 2, and repeat the process.
7. Iterate until you decide to stop

```
learn = Learner (dls, nn.Linear(28*28,1), metrics = batch_accuracy)
```

learn.fit()



Step 0 – load data

```
path = untar_data(URLs.MNIST_SAMPLE)
```

```
dls = ImageDataLoaders.from_folder(path)
```

Step 1 – initialize model

- Loading a custom-made model

```
learn = Learner(dls, simple_net, opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)
```

- Loading a model of an existing architecture

```
learn = cnn_learner(dls, resnet18, pretrained=False,
                    loss_func=F.cross_entropy, metrics=accuracy)
```

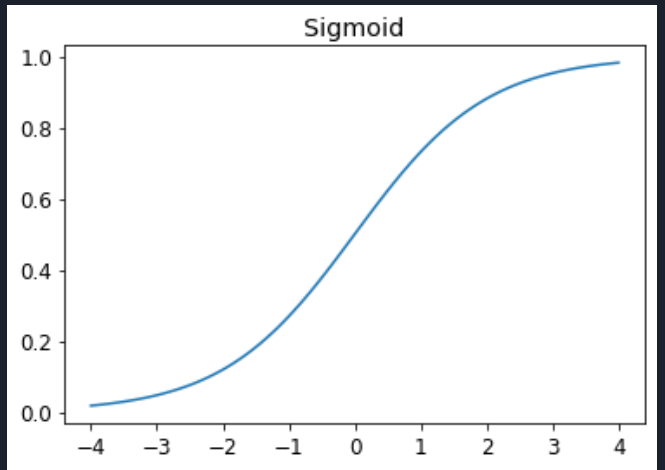
- What are the benefits of transfer learning?
- When is it appropriate?

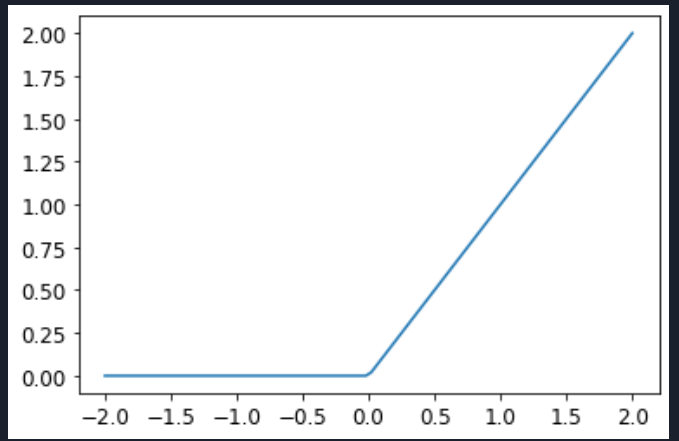
Step 2: Make a prediction

- Why include the bias term?
- Why include nonlinearity?
- What's the advantage of the RELU function?
- Why use sigmoid instead of the raw output?

```
def simple_net(xb):
    res = xb@w1 + b1
    res = res.max(tensor(0.0))
    res = res@w2 + b2
    return res
```

```
def sigmoid(x): return 1/(1+torch.exp(-x))
```





Step 3 – Calculate the loss function

- What is the difference between loss and metric?
- Why use cross-entropy as a loss function?

$$L = -[t \log y + (1 - t)\log(1 - y)]$$

Equals 0 when t = 0

Equals 0 when t = 1

Probabilistic interpretation of the loss function

- Output of the network as the probability that an input x belongs to class 1.

$$P(t=1|w,x) = y$$

$$P(t=0|w,x) = 1-y$$

- The likelihood function measures how well w predict the observed data and can be rewritten as the single equation:

$$P(t|w,x) = y^t(1-y)^{1-t} = e^{t\log y+(1-t)\log(1-y)}$$

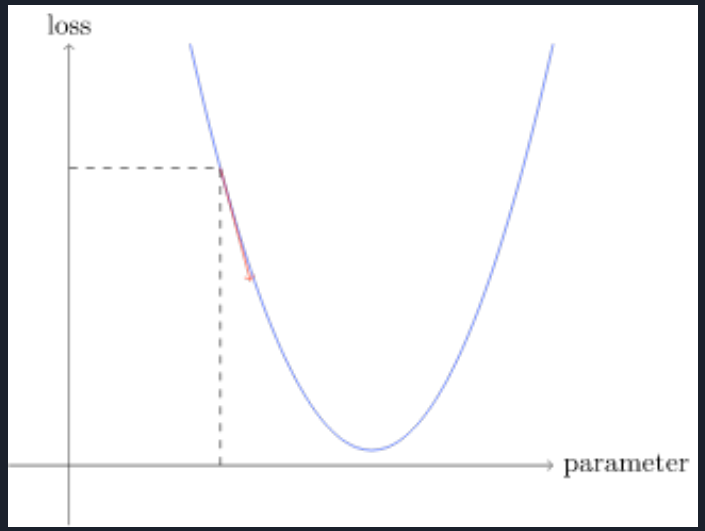
- The loss function can be interpreted as minus the log likelihood -log(P(D|w))
- Want to maximize the log likelihood (or minimize the negative log likelihood) of the correct class >> *Maximum Likelihood Estimation* (MLE).
- The loss function is the relative entropy (distance) between the empirical probability distribution of the targets and the probability distribution implied by the output of the model

Steps 4-5: Gradient descent

- Determines whether our loss will go up or down when we adjust our weights up or down. Gradients tell us how much we have to change each weight to make our model better.
- Why does the weight update formula have this form with the minus sign?

$$W_i \leftarrow W_i - \eta \frac{\partial L(W, X)}{\partial W_i}$$

```
def train_epoch(model, lr, params):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
        for p in params:
            p.data -= p.grad*lr
            p.grad.zero_()
```

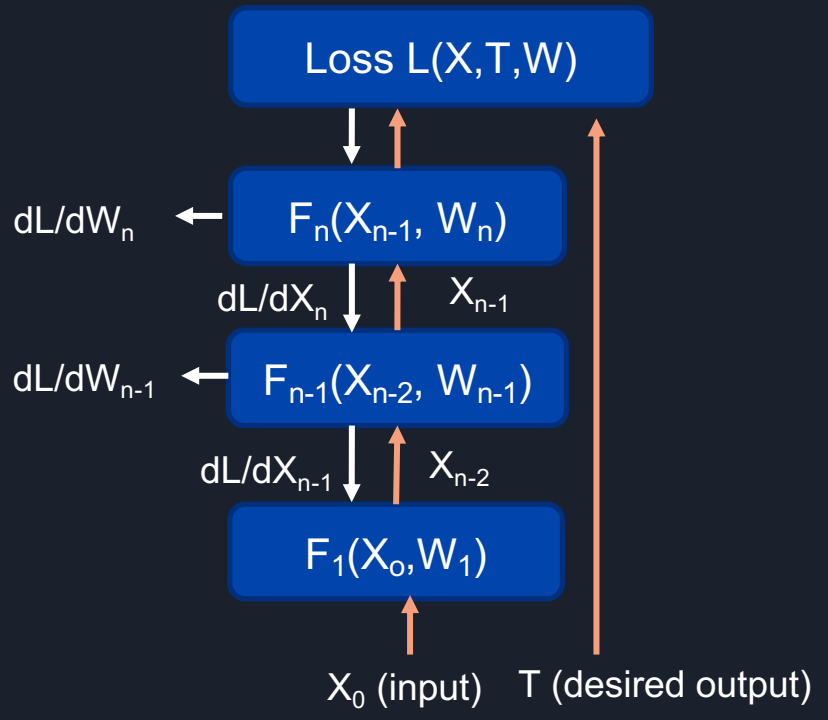


Backpropagation using the Chain Rule

Forward pass ↑

Back-propagate error ↓

$$\frac{dL}{dW_n} = \frac{dL}{dX_n}\frac{dX_n}{dW_n}$$



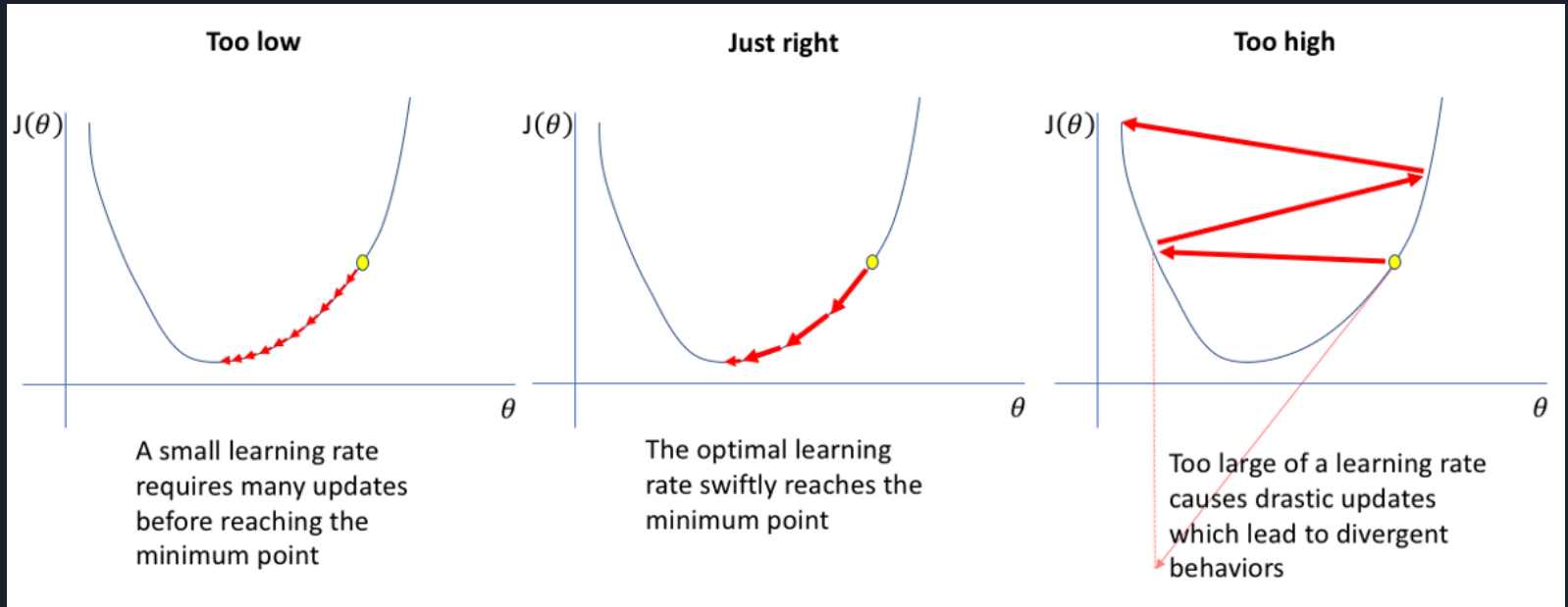
Steps 4-5: Gradient descent

- Updating the the weights based on the gradients is called an *optimization step*.
- We need to calculate the loss over one or more data items. How many should we use?

Steps 4-5

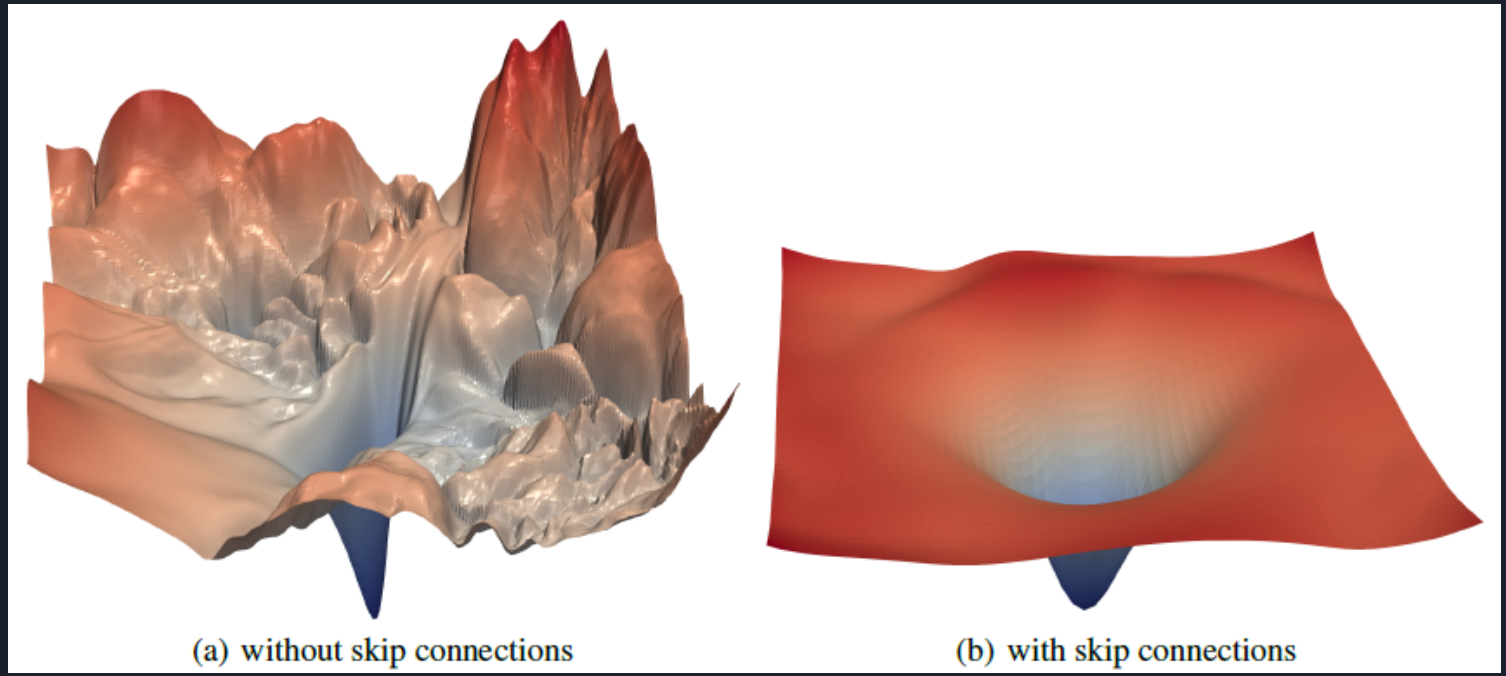
- Learning rate is one of the most important hyperparameters
- https://playground.tensorflow.org/

$$W_i \leftarrow W_i - \eta \frac{\partial L(W, X)}{\partial W_i}$$



Steps 4-5

- With a high-dimensional loss function, what can go wrong?



(a) without skip connections

(b) with skip connections

Loss function in high dimensions

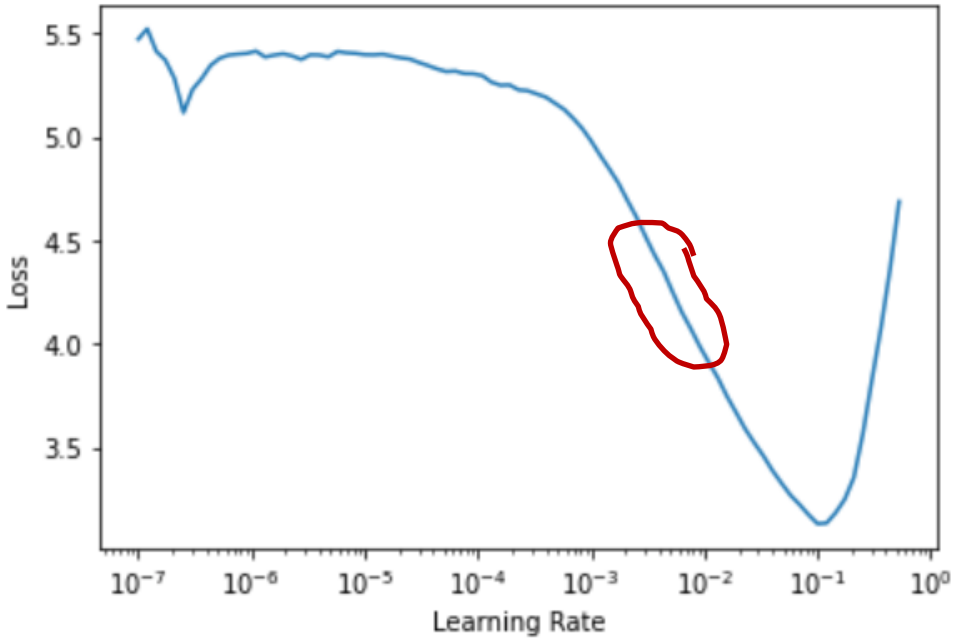
- *“the trainability of neural nets is highly dependent on network architecture design choices, the choice of optimizer, variable initialization, and a variety of other considerations.”*

- *“when networks become sufficiently deep, neural loss landscapes quickly transition from being nearly convex to being highly chaotic. This transition from convex to chaotic behavior coincides with a dramatic drop in generalization error, and ultimately to a lack of trainability.”*

- Visualizing the Loss Landscape of Neural Nets

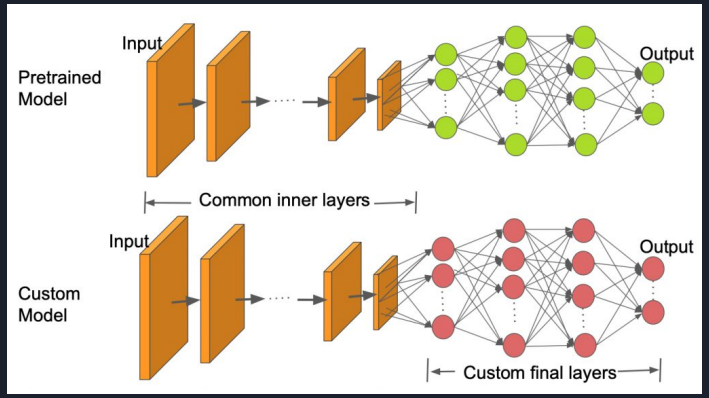
LR finder

```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
lr_min,lr_steep = learn.lr_find()
```



Fine tune – what's under the hood?

- Freezing/Unfreezing
- Fit one cycle
- Discriminative learning rates




```
def fine_tune(self:Learner, epochs, base_lr=2e-3, freeze_epochs=1, lr_mult=100,
              pct_start=0.3, div=5.0, **kwargs):
    "Fine tune with `freeze` for `freeze_epochs` then with `unfreeze` from `epochs` using
discriminative LR"
    self.freeze()
    self.fit_one_cycle(freeze_epochs, slice(base_lr), pct_start=0.99, **kwargs)
    base_lr /= 2
    self.unfreeze()
    self.fit_one_cycle(epochs, slice(base_lr/lr_mult, base_lr), pct_start=pct_start,
div=div, **kwargs)
```