Module 1:
Inductive reasoning in DL

feature engineering → to lower down the need for using all data-
(color histograms)

if we use softmax to convert scores to probabilities, the right loss function to use would be
cross-entropy

tensor → multi-dimensional matrices

each instance is a vector size of m, batch is of size [B x m]

each instance is a matrix of size WxH, our batch is [B x W x H]

each instance is a multi-channel matrix of size C x W x H our batch is [B x C x W x H]

Deep Learning

Since deep learning takes in inputs as raw with no dimensionality reduction and learns feature representations for that data.

- Representation Learning
- Neural Networks

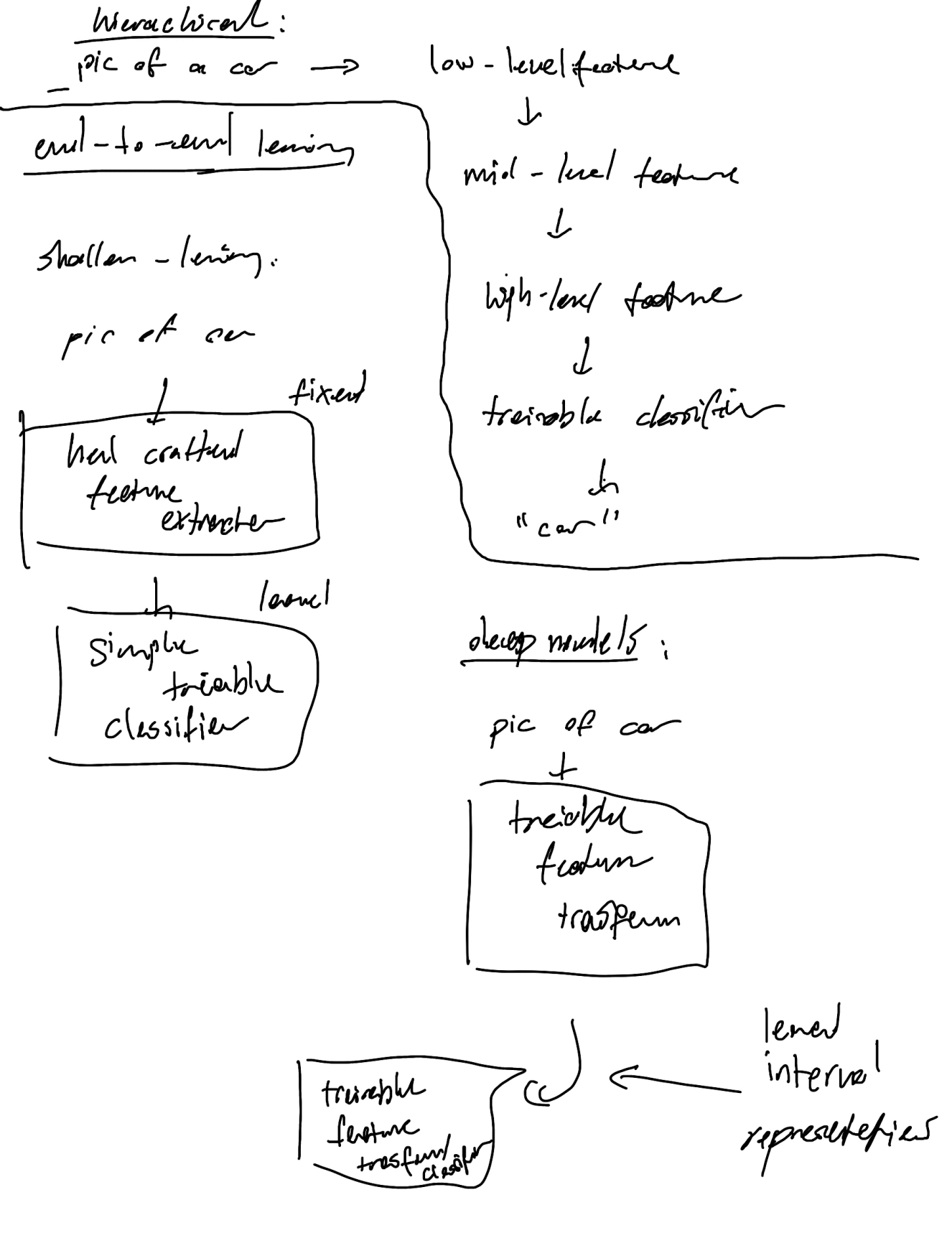
(Hierarchical) Compositionality:
cascade of non-linear transformations
multiple layers of representations

end-to-end learning
learning (goal-driven) representations
learning feature extraction

distributed representations
no single neuron "encodes" everything
groups of neurons work together

learn new features instead of being
told what the best feature is.

compose into a complex function


Hierarchical:
pic of a car
low-level features
mid-level features
high-level features
trainable classifier
"car"
end-to-end learning
shallow learning:
pic of car
fixed
hand crafted feature extractor
learned
simple trainable classifier
deep models:
pic of car
trainable feature transform
trainable feature transform classifier
learned internal representations

M1L2:

linear classifier

input ⇒ function of input -> loss function

input layer x | hidden layer W_1 | output layer W_2

$f(x, W_1, W_2) = 6(W_2\, 6(W_1 \cdot x))$

↑ sigmoid

three layer NN can hypothetically represent any function.

two layer NN can hypothetically represent any continuous function.

differentiability is important b/c of gradient descent.

backprop:

the outputs of the model is called forward

calculate the gradients for each module

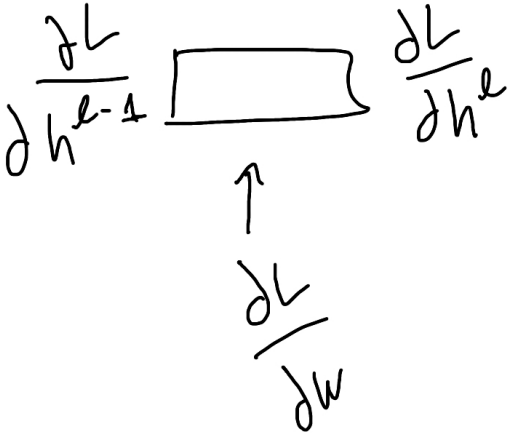
(backward pass)

backward pass → start at loss pass back through the layers modules and end at input layer (since it has no parameters

$h_1 \rightarrow h_2 \rightarrow h_3$ forward pass

(you also need to store intermediary outputs of all layers)

since we might need it for gradient calculation



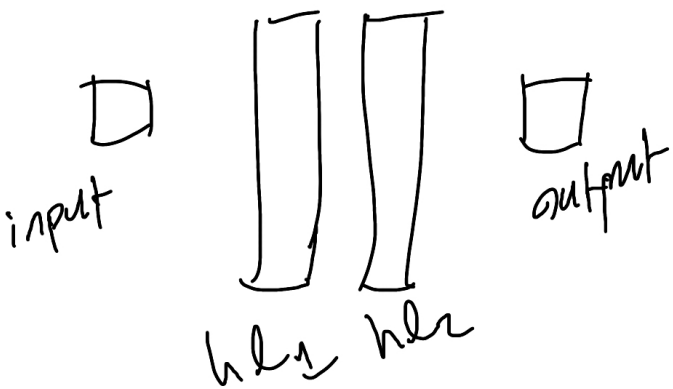
backward pass

$h_1 \leftarrow h_2 \leftarrow h_3$

ReLU has better gradient flow than sigmoid. Performed almost-wise

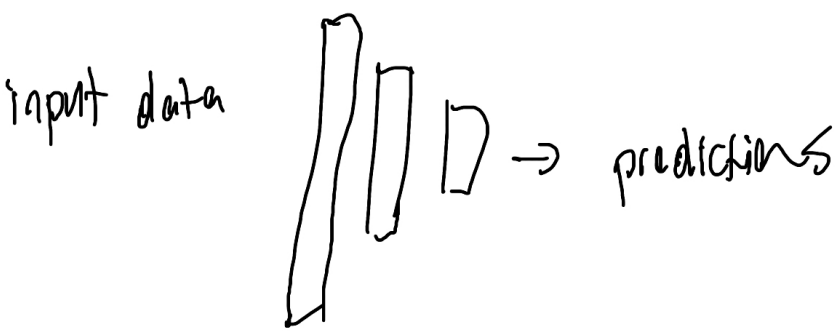
M123:

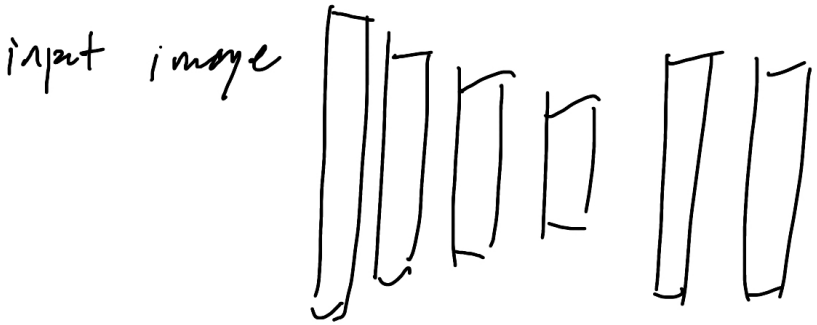
a network with two or more hidden layers is considered deep



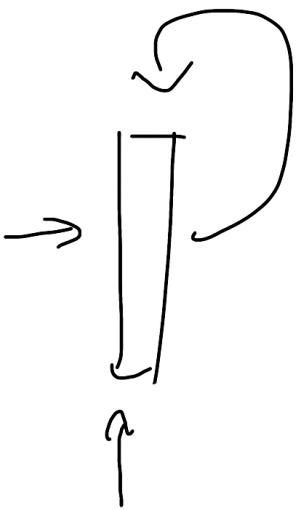
depth brings parameter efficiency dimensionality reduction

fully connected neural network





convolutional neural networks



designing architecture:

understanding your data is the first step.

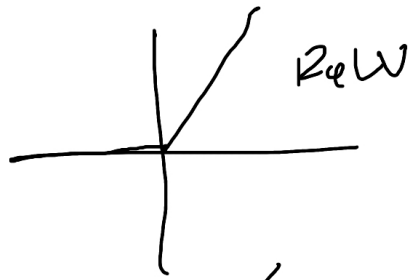
vanishing gradients ⇒ when gradients become flat.

tanh → centered / balanced.

features at the end of its derivatives are somewhat computationally heavy.

ReLU not centered like tanh but no saturation on positive end.

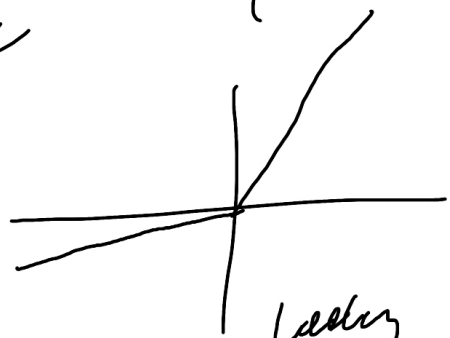
Leaky ReLU:



learnable parameters

no saturation on negative side

no dead neurons



Leaky ReLU

initializing values to constant values leads to a degenerate solution

- gradient will be the same
- all weights will be updated to same value.

common approach is small normally distributed random numbers

the deeper the network, the outputs of nodes become smaller, leading to lower standard deviations.

leads to small updates; larger lead to saturation.

Xavier initialization

simple initialization would:

Uniform $\left(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right)$

$n_j \rightarrow$ number of input nodes

$n_{j+1} \rightarrow$ number of output nodes

empirically well simpler version

$$N(0,1) \cdot \sqrt{\frac{1}{n_j}}$$

works for tanh, similar activations

ReLU:

(Xavier two?) $N(0,1) \cdot \sqrt{\frac{1}{n_j/2}}$

deep learning involves a lot eg complex, compositional, non-linear functions. so less landscape is non-convex a lot of intuition & rule of thumb used.

a good way to efficiently use the info is to take data in batches but this causes issues of high variance but it's unbiased. and how using descent.

dynamic learning rate

SGD can achieve similar results to

RMSProp

Adagrad

Adam

L1 → encourages sparcity.

L: $|y - Wx_i|^2 + \lambda \cdot |W|$

L2: L: $|y - Wx_i|^2 + \lambda \cdot |W|^2$

L1/L2 on weights encourage small values.

To prevent NN from overfitting
keep a 0.5 probability for each node
↳ allows you to check whether washing out a node helps.

in practice → it's a mask during test none is dropped-out. But you scale output by p.

$W_{test} : p \cdot W$

if let's say the "important features" are selected in dropout and the weights are updated in backprop, this technique will allow you to not rely on one particular feature

data augmentation → performing a range of transformations. it should also change the meaning of the data itself.

for example, image flipping, random cropping, color jitter

validation → it's for understanding architecture and what hyperparameter to use.

cross-validation → if you don't have a lot of data, a bit computationally expensive

check initial loss of small random weights $-\log(p)$ for cross entropy. where $p: 0.1$

or another example is to not do regularization and make sure loss goes up when added.

you can also test with a small dataset and see if your model can overfit this small amount of data before regularization.

tiny learn rate → too small of a learning rate

overfitting → validation loss starts to get worse after a while

underfitting → validation loss very close to training loss or both are high

validation loss here no regularization

hyper-parameters:

learning rate, weight decay

momentum, other more stable

batch norm & dropout not needed together.

learning rate should be changed proportionally to batch size (higher batch size higher learning rate)

TPR / FPR

tradeoff between the number of positive predictions and correctness of predictions.

TPR: $tp / tp + fn$

FPR: $fp / tp + tn$

accuracy: $(tp+tn) / (tp+tn+fn+fp)$

Missing data imputation

for numerical data:

mean, mode, most frequent, zero, constant

for categorical data:

hot-deck imputation, k-NN, deep learned embeddings