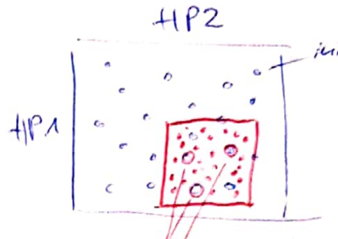
Hyperparameter tuning

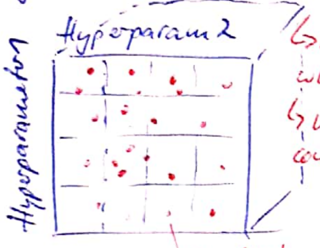
1st α ← most important
2nd β = ~0.9 should be ok
$\beta_1, \beta_2, \varepsilon \rightarrow 0.9,\ 0.999,\ 10^{-8}$ — usually not varied much
3rd # layers
2nd # hidden units
3rd learning rate decay
2nd mini-batchsize

Coarse to fine sampling



these worked best, so randomly sample around then → add finer sample points

Try random values: Don't use a grid!



↳ reason: don't know up front which parameter is most important
↳ will end up with more different configurations for α than with a grid
⇒ more richly exploring! (like DoE)

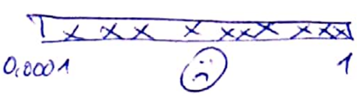
Using an appropriate scale to pick hyperparameters (HP)

reasonable picking at uniform random → not true for all HP!

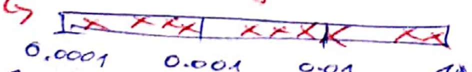
→ $n^{[l]} = 50, \dots, 100$ →

→ # layers L : 2 − 4 → 2, 3, 4

example α : 0,0001, …, 1 →



rather go for log-scale
↳



10^a 10^b

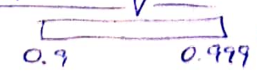
$a = \log_{10} 0.0001$ $b = \log_{10} 1$
$a = -4$ $b = 0$

⇒ $r \in [a, b]$ $(-4, 0)$

Python: `r = -4 * np.random.rand()` ← $r \in [-4, 0]$
$\alpha = 10^r$ ← $10^{-4}, \dots, 10^0$

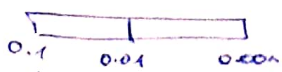
HPs for exponentially weighted averages

$\beta = 0.9 \ldots\ldots 0.999$
$\underbrace{\quad}_{10}$ $\underbrace{\quad}_{1000}$



→ better not sample uniform randomly
→ but with log scale!

$1-\beta = 0.1 \ldots 0.001$



10^{-1} 10^{-3}

$r \in [-3, -1]$
$1-\beta = 10^r$
$\beta = 1 - 10^r$

Why is sampling on linear scale a bad idea?
→ when β close to 1 the sensitivity of the results changes even for very low changes in β.

$\beta: 0.9 \rightarrow 0.9005$ } ~10 values
$\beta: 0.999 \rightarrow 0.9995$ } ~1000 values   ~2000

$\frac{1}{1-\beta}$! very sensitive to changes to β

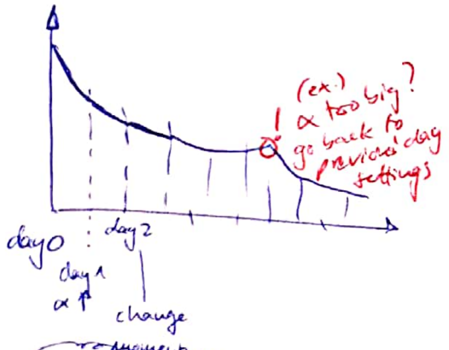
Pandas vs Caviar

Re-test HPs occasionally

Idea → Code → Experiment

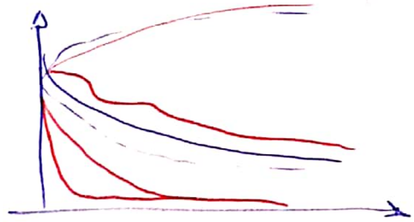
- NLP, Vision, Speech, Ads, logistics
- Intuitions do get stale
  Re-evaluate occasionally

Babysitting one model
(one model at a time)



"Panda"
one child at a time

Training many models in parallel



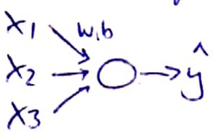
depends also on computational power!

"Caviar"
many children at a time

Batch normalization → makes HP search easier and the ANN more robust

Normalizing activations in a network → applies not only to inputs but also to deeper layers!

↳ Normalizing inputs to speed up learning

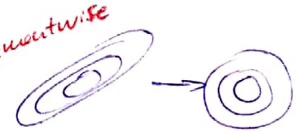


$\mu = \frac{1}{m}\sum_i x^{(i)}$

$x = x - \mu$

$\sigma^2 = \frac{1}{m}\sum_i x^{(i)2}$ / elementwise

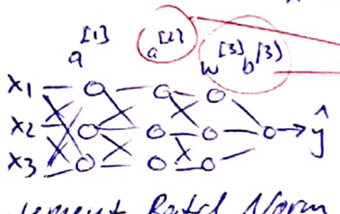
$x = \frac{x}{\sigma^2}$



might not want hidden units to be forced to mean 0 and variance 1

e.g.

don't want values to be clustered here in the linear regime but want to take advantage of the non-linearity



would be nice to normalize mean and variance to make training of $w^{[3]}, b^{[3]}$ more efficient

Q: Can we normalize $a^{[2]}$ so as to train $w^{[3]}, b^{[3]}$ faster?

↳ Normalize $z^{[2]}$ (before activation but also possible to normalize after activation)

Implement Batch Norm

Given some intermediate values in NN $\underbrace{z^{(1)}, \dots, z^{(m)}}_{(z^{[l](i)})}$

$\mu = \frac{1}{m}\sum_i z^{[l](i)}$

$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2$

$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ ⇒ to have mean 0 and standard unit variance

add ε for numerical stability if $\sigma^2 = 0$ for some estimate

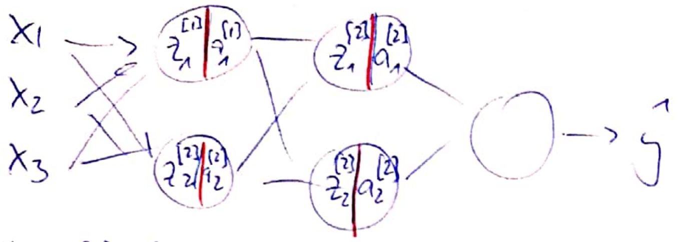
→ but sometimes it might be better for some hidden layers to have a different distribution:

$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$ ← learnable params of our model
↳ update in GD just as W, b

if $\gamma = \sqrt{\sigma^2 + \varepsilon}$, $\beta = \mu$ } would invert $z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$ and $\tilde{z}^{(i)} = z^{(i)}$

↳ use $\tilde{z}^{[l](i)}$ instead of $z^{[l](i)}$ for later calculation

Fitting Batch Norm into a NN



In tensorflow: use BN via tf.nn.batch-normalization

usually

$$X \xrightarrow{W^{[1]},\, b^{[1]}} z^{[1]} \rightarrow a^{[1]}$$

with batch norm:

$$X \xrightarrow{W^{[1]},\, b^{[1]}} z^{[1]} \xrightarrow[\text{Batch Norm (BN)}]{\beta^{[1]},\, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{W^{[2]},\, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]},\, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$$

→ mean and variance normalized

Parameters in NN: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]},$
$\beta^{[1]}, \gamma^{[1]}, \dots \beta^{[L]}, \gamma^{[L]}$ } Use GD $\boxed{\beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}}$

In practice BN is usually applied with mini-batches

Working with mini-batches

$$X^{\{1\}} \xrightarrow{W^{[1]},\, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]},\, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow \dots$$

then

$$X^{\{2\}} \rightarrow z^{[2]} \xrightarrow[BN]{\beta^{[1]},\, \gamma^{[1]}} \tilde{z}^{[2]}$$

$X^{\{3\}}$

$\vdots$

$X^{\{T\}}$

Parameter: $W^{[l]}, \cancel{b^{[l]}}, \beta^{[l]}, \gamma^{[l]}$

parameter can be eliminated when using batch norm

$$z^{[l]} = W^{[l]} a^{[l-1]} + \cancel{b^{[l]}}$$

normalize μ 0 std var.

compute means and subtract out the mean so adding a constant to all the mini-batch to all examples in a mini-batch doesn't change anything because added constant gets cancelled out

$z^{[l]} = W^{[l]} a^{[l-1]}$

$z^{[l]}_{norm}$

$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$

use to decide what is the mean of $\tilde{z}^{[l]}$

- because BN zeros out the mean of $z^{[l]}$, $b^{[l]}$ does not need to be kept and is sort of replaced by $\beta^{[l]}$ → affects shift/bias
- dimension of $z^{[l]} \rightarrow (n^{[l]}, 1)$
  $\beta^{[l]}, \gamma^{[l]} \rightarrow (n^{[l]}, 1)$

Implementing gradient descent

assumption: mini-batch is used

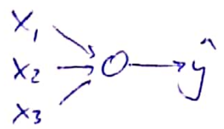
for $t = 1 \ldots$ num mini-batches

- compute FP on $X^{\{t\}}$
  - In each layer use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
  - → ensures the mean and variance normalized
- Use backprop to compute $dW^{[l]}$, ~~$db^{[l]}$~~, $d\beta^{[l]}$, $d\gamma^{[l]}$ ($\tilde{z}$ is used)
- Update parameters $W^{[l]} := W^{[l]} - \alpha dW^{[l]}$
  - $\beta^{[l]}$:
  - $\gamma^{[l]}$:

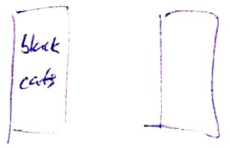
→ works also with GD momentum, RMSprop, Adam

Why does BN work?

→ makes "later" hidden units more robust to changes in "earlier" hidden units
↳ weakens coupling between layers and so allows each layer to learn by itself more independently of other layers
→ speeding up learning!



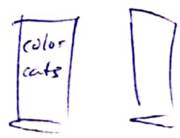
Cat $y=1$ Non-Cat $y=0$



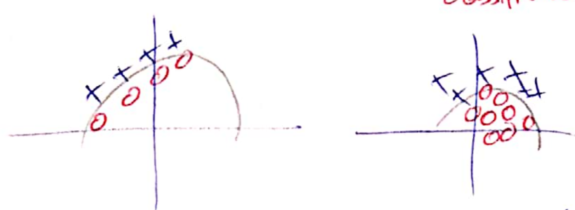
trained only of images of black cats, now want to apply on images with colored cats

classifier will fail

$y=1$ $y=0$



$\boxed{\text{data distribution changing} \rightarrow \text{covariate shift}}$



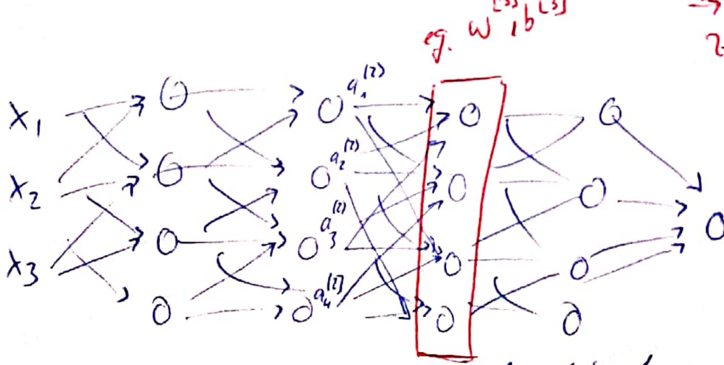
Would not expect model trained on left data perform well on right data

$X \xrightarrow{\text{mapping}} y$ if distribution of x changes then we might need to re-train our model (ground truth function shift)

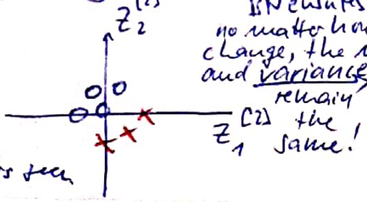
How does this apply to a NN?

BN → reduces problem of* input values changing

e.g. $W^{[3]}, b^{[3]}$

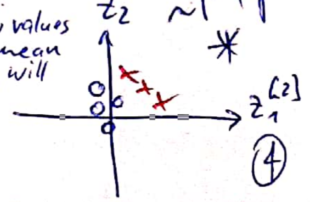


→ NN adapting all params
↳ when e.g. $W^{[2]}, b^{[2]}$ change, also W, b in $l=3$ change
↳ suffering from covariate shift!
→ BN reduces the amount that distribution of hidden unit values shifts around!



BN ensures: no matter how values change, the mean and variance will remain the same!

$z_2^{[2]} \sim \beta^{[2]}, \gamma^{[2]}$ *



* limits amount to which updates of params in earlier layers can affect the distribution of values that the next layers see

Batch Norm as regularization

→ Each mini-batch is scaled by mean/variance computed on just that mini-batch

→ This adds noise to the values of $z^{[l]}$ within that mini-batch.
Similar to dropout, it adds some noise to each hidden layer's activations

→ This has slight regularization effect
↳ but not the idea (BN is no regularization method)

↳ BN scaling by σ^2 noise
additive noise by subtracting the μ

→ If using a bigger mini-batch size noise is reduced but also regularization effect is reduced

Batch Norm at test time

→ BN processes data on mini-batch at a time

→ but at test time we might need to process the examples one at a time

↳ adapt NN to do that

$$\boxed{\begin{aligned}\mu &= \frac{1}{m}\sum_i z^{(i)} \\ \sigma^2 &= \frac{1}{m}\sum_i (z^{(i)}-\mu)^2\end{aligned}}$$

m — examples per mini-batch, not whole training set!

— computed on mini-batch

$$z^{(i)}_{norm} = \frac{z^{(i)}-\mu}{\sqrt{\sigma^2+\varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

need new way to get μ, σ →

μ, σ^2: estimate via exponentially weighted average (across mini-batches)

e.g. layer $l = \ldots$

$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}$

$\cancel{\mu^{[l]}}$ $\cancel{\mu^{[l]}}$ $\cancel{\mu^{[l]}}$

$\mu^{\{1\}[l]}$ $\mu^{\{2\}[l]}$ $\mu^{\{3\}[l]}$ $\rightsquigarrow \mu$

θ_1 θ_2 θ_3 σ^2

$\sigma^{2\{1\}[l]}$ $\sigma^{2\{2\}[l]}$ ... → running average of μ, σ^2 of each layer as we train NN across multiple mini batches

at test time calculate $z_{norm} = \frac{z-\mu}{\sqrt{\sigma^2+\varepsilon}}$ instead of

↓ use exp. weight. avg.

↳ $\tilde{z} = \gamma z_{norm} + \beta$

~~→ μ, σ^2 are comp. on batch~~

→ estimate μ, σ^2 from training set

→ in practice implement exp. weigh. avg. to keep track of μ, σ^2 (via running average)

Multiclass-classification

Softmax Regression

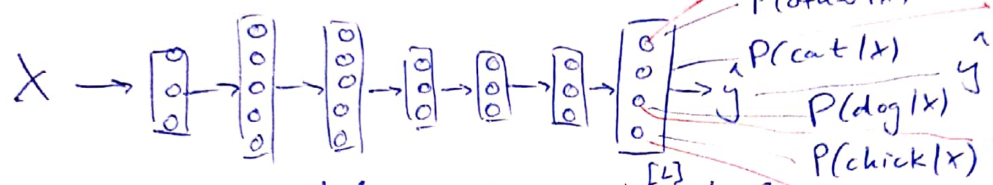
recognizing cats, dogs and baby chicks, others

chick	cat	dog	koala	chick	dog	koala	cat
3	1	2	0	3	2	0	1

$C = \#\text{classes} = 4$ $(0, \dots, 3)$

$\frac{e^5}{176.3} \approx 0.842$

$\frac{e^2}{176.3} = 0.042$



$\hat{y}$ is (4,1) } should sum to 1

$\frac{e^{-1}}{176.3} = 0.002$

$\frac{e^3}{176.3} = 0.114$

Standard way is Softmax layer $n^{[L]} = 4 = C$

$$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$$

Activation function:

temporary variable $\underset{(4,1)}{t} = e^{(z^{[L]})}$ (element-wise exponentiation)

$$\hat{y} \Rightarrow \underset{(4,1)}{a^{[L]}} = \frac{e^{(z^{[L]})}}{\sum_{i=1}^{4} t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{i=1}^{4} t_i}$$

4 classes

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix} \cdot \sum_{i=1}^{4} t_i = 176.3$$

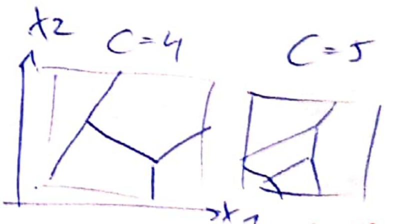
$$a^{[L]} = \frac{t \text{ (vector)}}{176.3}$$

$$\hat{y} = a^{[L]} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

$\Sigma = 1$

$$\Rightarrow \underset{(4,1)}{a^{[L]}} = g^{[L]}(\underset{(4,1)}{z^{[L]}})$$

this activation function inputs a (4,1) vector and outputs a (4,1) vector



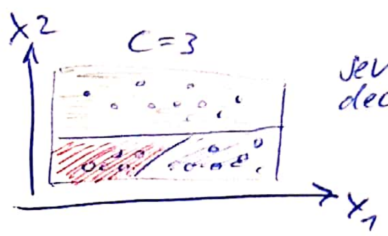
Softmax examples

no hidden layer

$x_1, x_2 \rightarrow [\,\circ\,\circ\,\circ\,] \rightarrow \hat{y}$ $\quad z^{[1]} = W^{[1]} x + b^{[1]}$

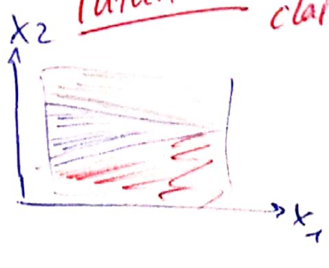
$$a^{[1]} = \hat{y} = g(z^{[1]})$$

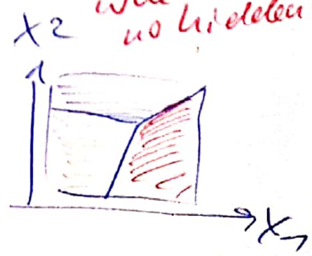
with hidden layers also non-linear decision boundaries!!!



several linear decision boundaries

Intuition: Decision boundaries between classes will be linear when there is no hidden layer!





Training a softmax classifier

example

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \rightarrow a^{[L]} = g^{[L]}(z^{[L]}) = \begin{bmatrix} e^5/(e^5+e^2+e^{-1}+e^3) \\ e^2/ \quad " \\ e^{-1}/ \quad " \\ e^3/ \quad " \end{bmatrix} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

Softmax	"hardmax"
$\begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$
↓ more gentle mapping	↓ hard mapping

If $C=2$, softmax reduces to logistic regression

$\Rightarrow$ ex. $a^{[L]} = \begin{bmatrix} 0.842 \\ 0.158 \end{bmatrix}$ (actually redundant! just compute one and get the other one automatically as they have to add up to 1)

Softmax regression generalizes logistic regression to C classes

Loss function

$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ – cat NN is currently outputting $a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$ – doing not too well

$y_1 = y_3 = y_4 = \underline{\underline{0}}$

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log(\hat{y}_j) \qquad \Big|\ J(w^{[1]}, b^{[1]}, \ldots) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

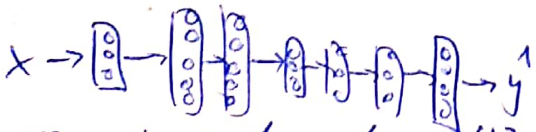
$= -y_2 \log \hat{y}_2$ (all except "2" were 0) Use GD to minimize J

to make $\mathcal{L}(\hat{y}, y)$ small we need to make $-\log \hat{y}_2$ small, so we make $\hat{y}_2$ big!

$$Y = [y^{(1)}, y^{(2)}, \ldots y^{(m)}] \qquad \hat{Y} = [\hat{y}^{(1)}, \ldots \hat{y}^{(m)}]$$

$$= \begin{bmatrix} 0 & 0 & 1 & \\ 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \\ 0 & 0 & 0 & \end{bmatrix}_{(4,m)} \qquad = \begin{bmatrix} 0.3 & \\ 0.2 & \\ 0.1 & \cdots \\ 0.4 & \end{bmatrix}_{(4,m)}$$

Gradient Descent with softmax



FP is straight forward $\underset{(4,1)}{z^{[L]}} \rightarrow a^{[L]} = \hat{y} \rightarrow \mathcal{L}(\hat{y}, y)$

BP: $\underset{(4,1)}{dz^{[L]}} = \hat{y} - y$

$\frac{\partial J}{\partial z^{[L]}}$

Deep Learning frameworks

- Caffe/Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Choosing DL Framework

- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)

TensorFlow (1) / Programming assignment

ex. $J(w) = \boxed{w^2 - 10w + 25}$

$\uparrow (w-5^2)$

Create & use sessions

method 1

```
sess = tf.Session()
result = sess.run(..., feed_dict = {...})   w=5
sess.close()
```

method 2

```
with tf.Session() as sess:
```

feed data: result = sess.run(tf.sigmoid(x), feed_dict = {x : z})
for running a session (closes session automatically)

dictionary here

with arrays as inputs ⟶ result = sess.run(cost, feed_dict = {z: logits, y=labels})

→ Create tensors/variables that are not executed yet

→ Define operations between the tensors
tf.matmul / tf.add

→ Initialize tensors

→ Create a session

→ run the session

"one hot" encoding

result vector

$$y = \{1\ 2\ 0\ 3\ 2\} \longrightarrow \begin{array}{c} C=3 \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{class 0} \\ \text{class 1} \\ \text{class 2} \\ \text{class 3} \end{array}$$

↳ tf.one_hot(labels, depth, axis)

↳ tf.one_hot(indices=labels, depth=C, axis=0)

initialize to zero and one

```
tf.ones(shape)
tf.zeros(shape)
```

tensor flow constant "y" with value 5

```
tf.constant(5, name = "y")
```

```
a = tf.placeholder(tf.float32, name = "a")
x = tf.variable(a + 3, name = "x")
```

constant or placeholder

8

Tensor Flow / Programming assignment sign language!

→ Create a computation graph
→ Run the graph

train set = 1080 pictures (64×64 pixels) ↳ 64×64×3 - RGB
test set = 120 pictures
C = 6 → hand signals from 0 to 5

→ flatten the image + nomalize by 255

X_train_flatten = X_train_orig.reshape(X_train_orig.shape[0], -1).T
↳ same for Y and test...

→ Create placeholders → example: X = tf.placeholder(tf.float32, shape = (..., ...), name = "X") or [..., ...]

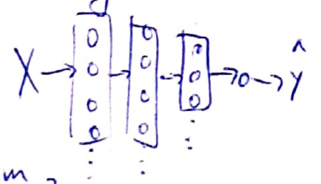
→ Initialize parameters → example: W1 = tf.get_variable("W1", [25, 12288], initializer = tf.contrib.layers.xavier_initialization)

→ Forward propagation → outputs no cache here...
example: Z2 = tf.add(tf.matmul(W2, A1), b2)
A2 = tf.nn.relu(Z2)

→ Compute cost → input(Z3, Y) → logits = tf.transpose(Z3)
labels = tf.transpose(Y)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = logits, labels = labels))

→ Backpropagation + parameter updates
→ after cost, create object optimizer, called together with cost in tf.session
e.g. optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(cost)
or .AdamOptimizer. ...
to do optimization:
→ c = sess.run([optimizer, cost], feed_dict = {X: mini_batch_X, Y: mini_batch_Y})

→ Building the model



$X = n_x \overset{m}{[\,|\,|\,|\,|\,]}$