Artificial Intelligence and Machine Learning

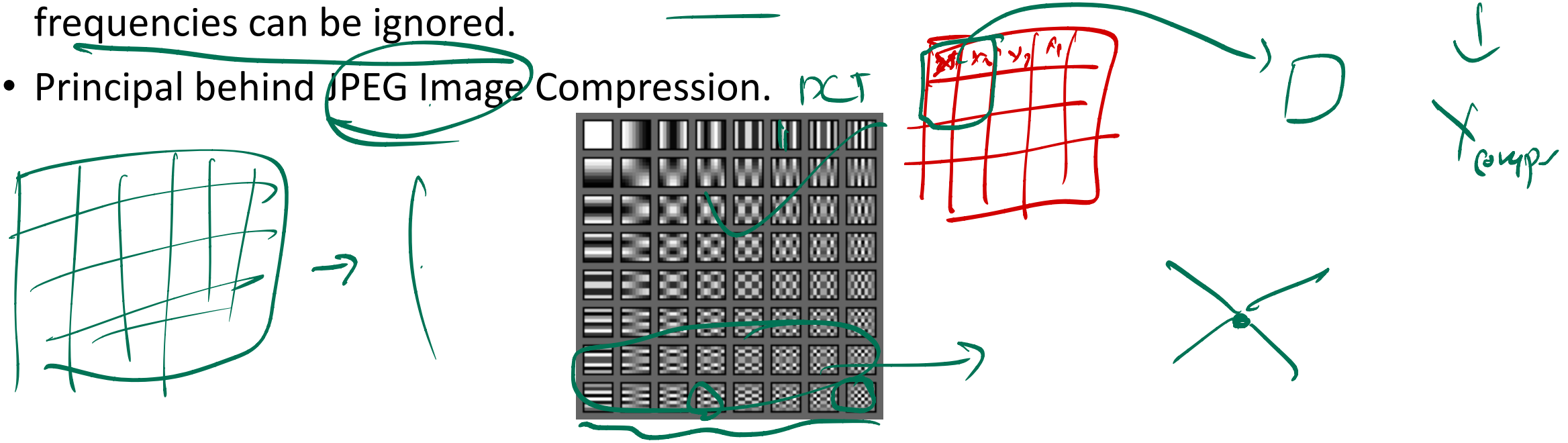
Applications

Lecture Outline

- Linear Regression Tutorial & Playground
- Image Compression
- Auto-Encoders
- Optimizers
- Logistic Regression Interactive Demo
- Neural Network Regression Tutorial & Playground

Image Compression with Linear Regression

- An image can be transformed into the frequency domain and represented as a combination of some basic components.
- Cosine Basis Functions and Discrete Cosine Transforms (DCT) can enable this.
- The human eye is most sensitive to low frequencies. Therefore, most of the high frequencies can be ignored.
- Principal behind JPEG Image Compression.



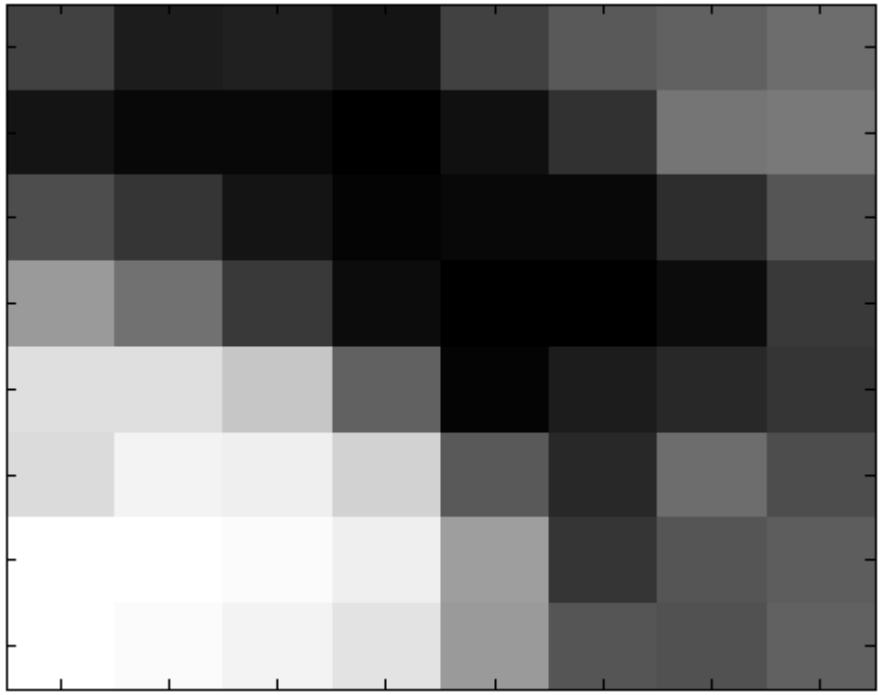
https://bengarney.com/2016/06/25/video-conference-part-2-joint-photographic-hell-for-beginners/

Image Compression with Linear Regression

- Using linear regression, we can learn the weight of each of these cosine basis.
- We only need to store the weights of the basis frequencies.
- The image is reconstructed using these weights.

Image Compression

8 x 8 Pixels



Image

Slide based on Discrete Cosine Transform and Image Compression by Thomas Buck et. Al.

Image Compression

- Gray-Scale Example:
- Value Range 0 (black) --- 255 (white)

63	33	36	28	63	81	86	98
27	18	17	11	22	48	104	108
72	52	28	15	17	16	47	77
132	100	56	19	10	9	21	55
187	186	166	88	13	34	43	51
184	203	199	177	82	44	97	73
211	214	208	198	134	52	78	83
211	210	203	191	133	79	74	86

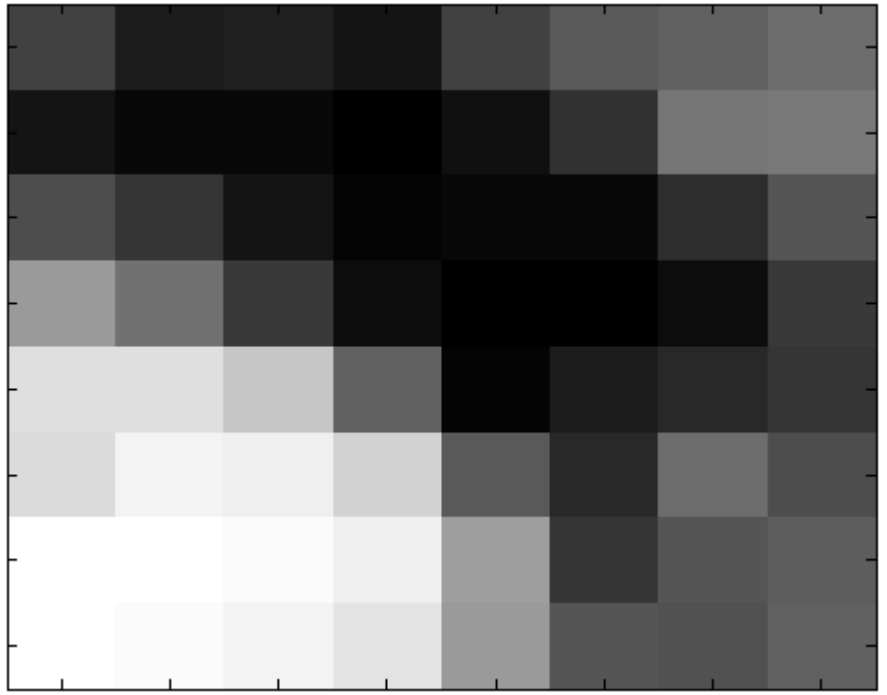
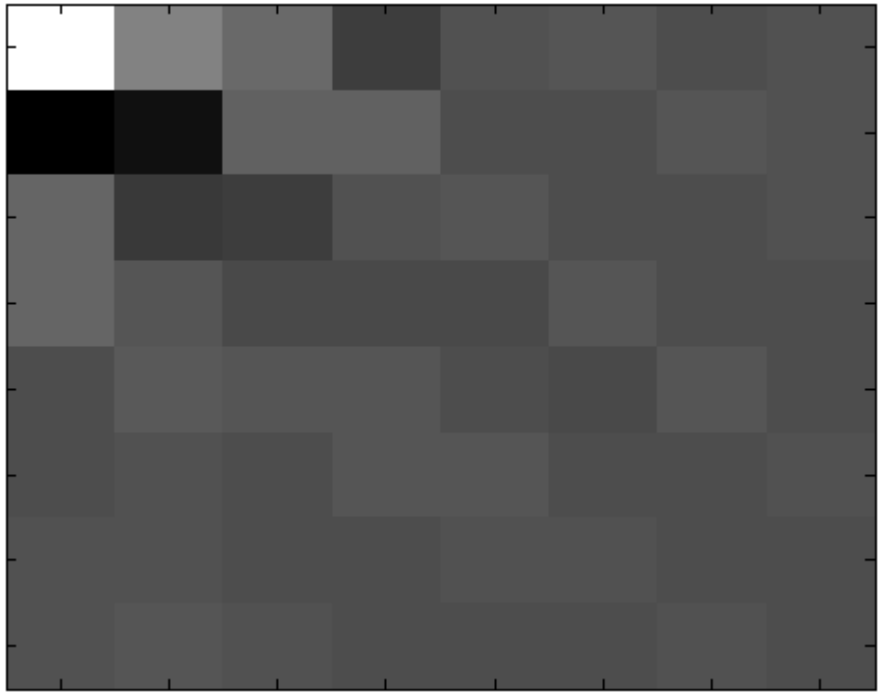


Image Compression

- 2D-DCT of matrix

Numbers are coefficients of polynomial

-304	210	104	-69	10	20	-12	7
-327	-260	67	70	-10	-15	21	8
93	-84	-66	16	24	-2	-5	9
89	33	-19	-20	-26	21	-3	0
-9	42	18	27	-7	-17	29	-7
-5	15	-10	17	32	-15	-4	7
10	3	-12	-1	2	3	-2	-3
12	30	0	-3	-3	-6	12	-1

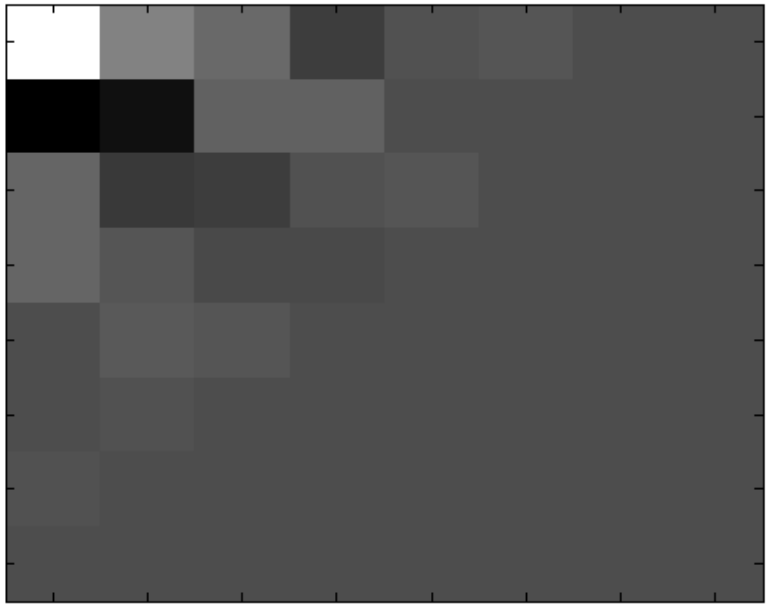


Slide based on Discrete Cosine Transform and Image Compression by Thomas Buck et. Al.

Image Compression

- Cut the least significant components

-304	210	104	-69	10	20	-12	0
-327	-260	67	70	-10	-15	0	0
93	-84	-66	16	24	0	0	0
89	33	-19	-20	0	0	0	0
-9	42	18	0	0	0	0	0
-5	15	0	0	0	0	0	0
10	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

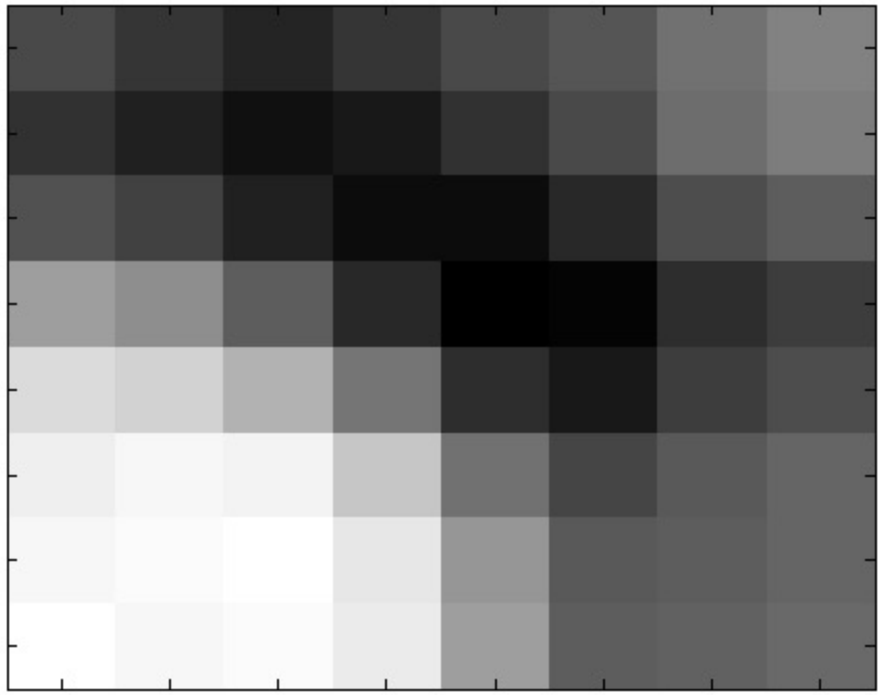


As you can see, we save a little over half the original memory.

Reconstructing the Image

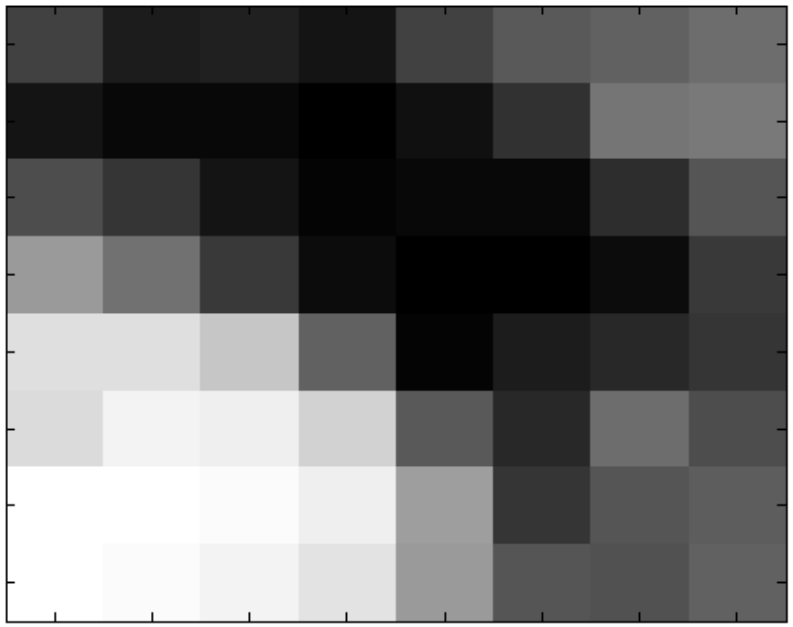
- New Matrix and Compressed Image

55	41	27	39	56	69	92	106
35	22	7	16	35	59	88	101
65	49	21	5	6	28	62	73
130	114	75	28	-7	-1	33	46
180	175	148	95	33	16	45	59
200	206	203	165	92	55	71	82
205	207	214	193	121	70	75	83
214	205	209	196	129	75	78	85

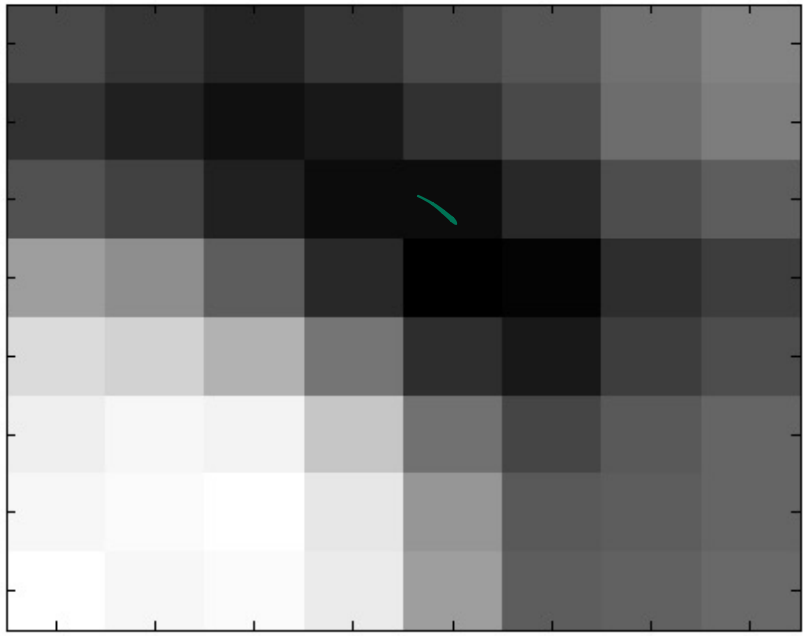


Can You Tell the Difference?

Original



Compressed



Slide based on Discrete Cosine Transform and Image Compression by Thomas Buck et. Al.

Image Compression

Original

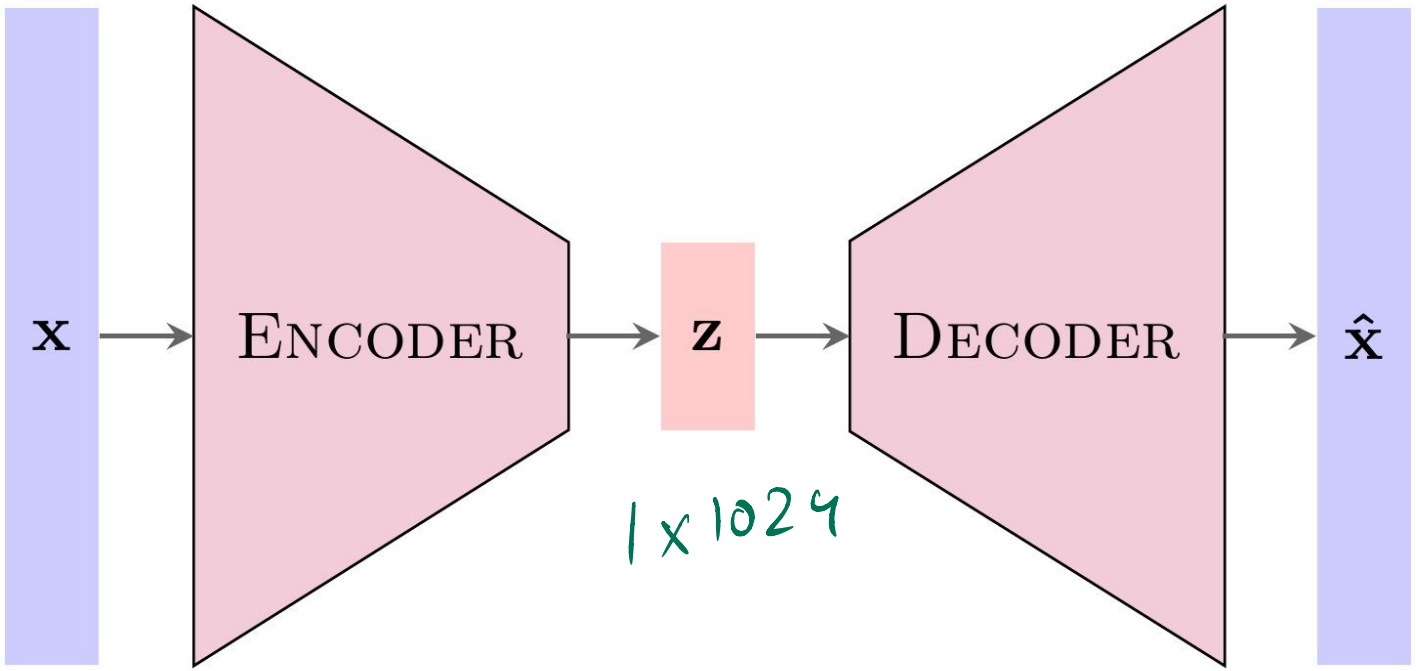
Compressed

Slide based on Discrete Cosine Transform and Image Compression by Thomas Buck et. Al.

Applications – AutoEncoders

- Autoencoders are a type of neural networks where the input is also the output.
- They come under unsupervised learning and there are no labels involved.
- An autoencoder consists of two parts: encoder and decoder.
- The idea here is that you take a higher dimensional input, project it into a lower dimensional space and then project it back into the input space.

Applications – AutoEncoders



https://github.com/JamesAllingham/LaTeX-TikZ-Diagrams

Applications – AutoEncoders

- The autoencoder model tries to minimize the reconstruction error (RE).
- Typically, mean squared is used as the loss function for autoencoders.
- The objective is to minimize the following:

$$L(x, \hat{x}) = \frac{1}{N} \sum_{i=1}^{N} ||x_i - \hat{x}_i||^2$$

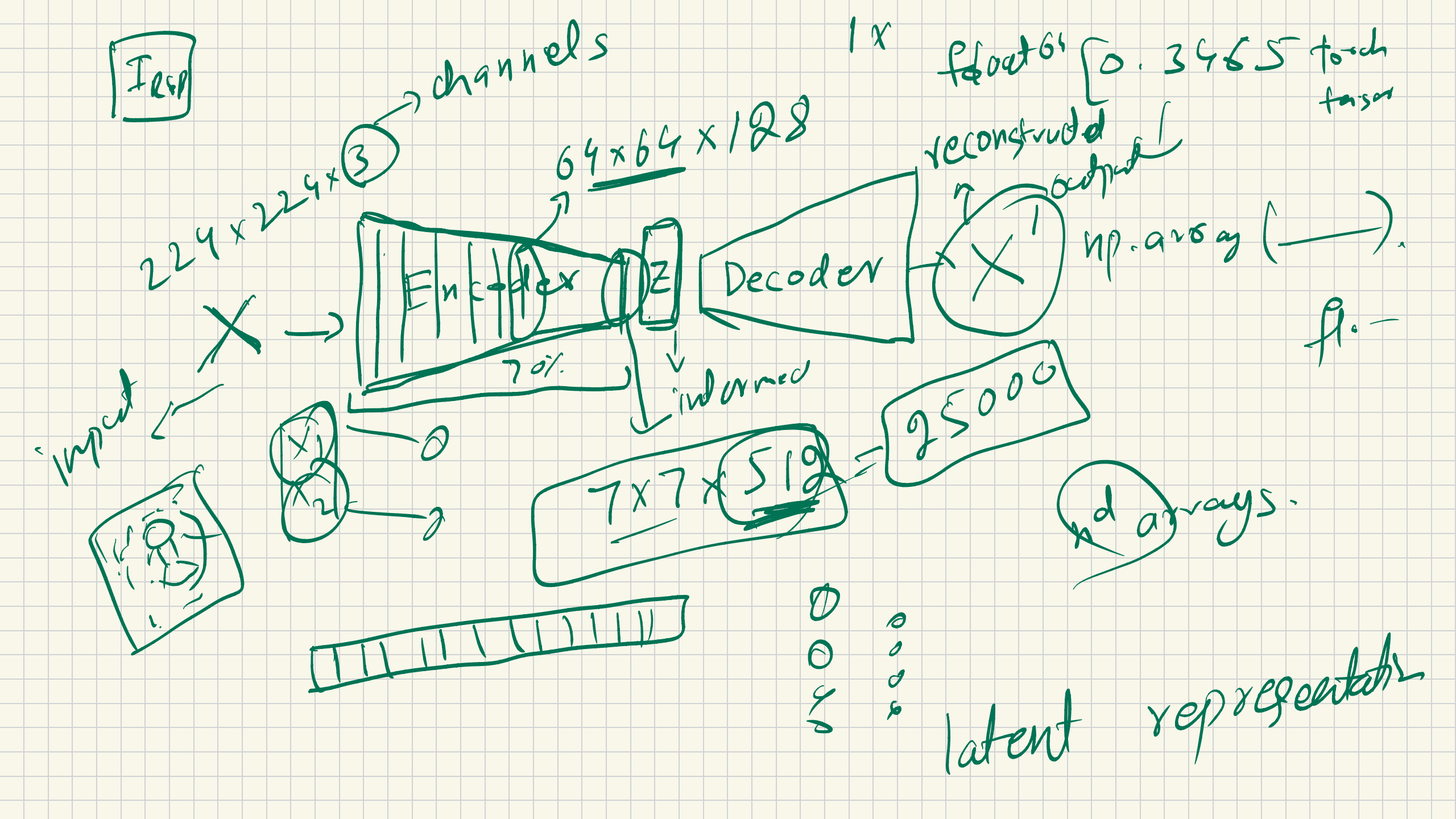
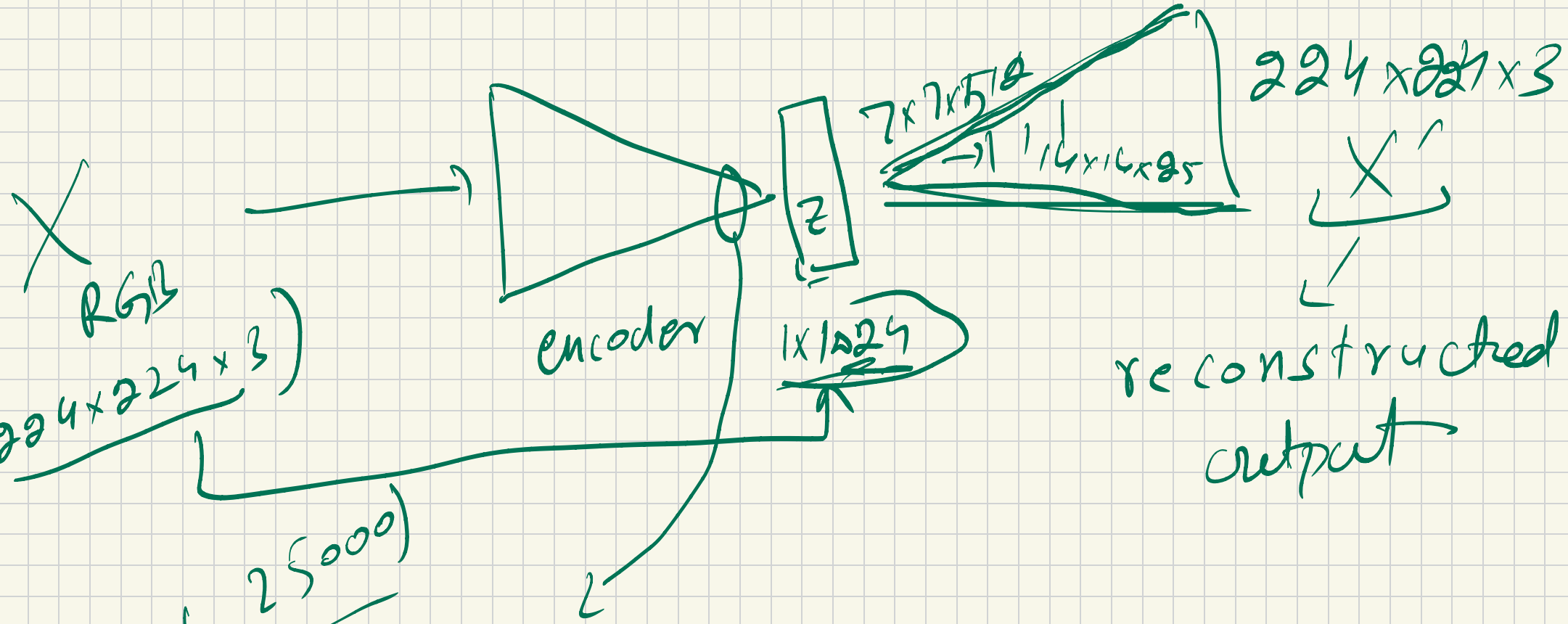


Image
channels
3
224 x 224 x
64 x 64 x 128
Encoder
Z
Decoder
reconstructed
output
X'
np.array(
float64
0.3465
torch
tensor
input
X
70%
"informed"
X1
X2
0
7 x 7 x 512
25000
1d arrays.
latent representation



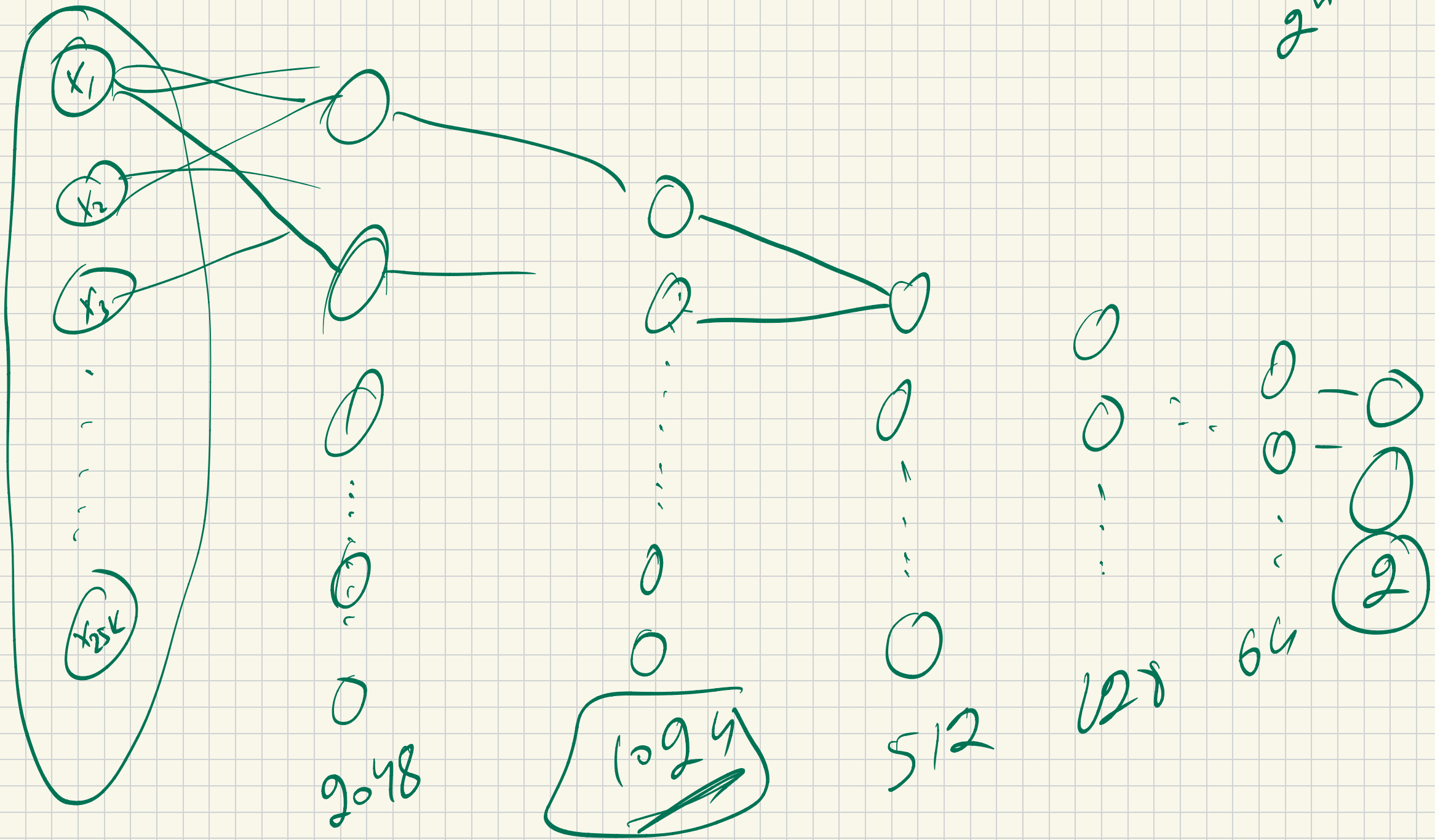
x.view(-1, 25000)

1×25000 = nn.flatten(x)

7×7×512

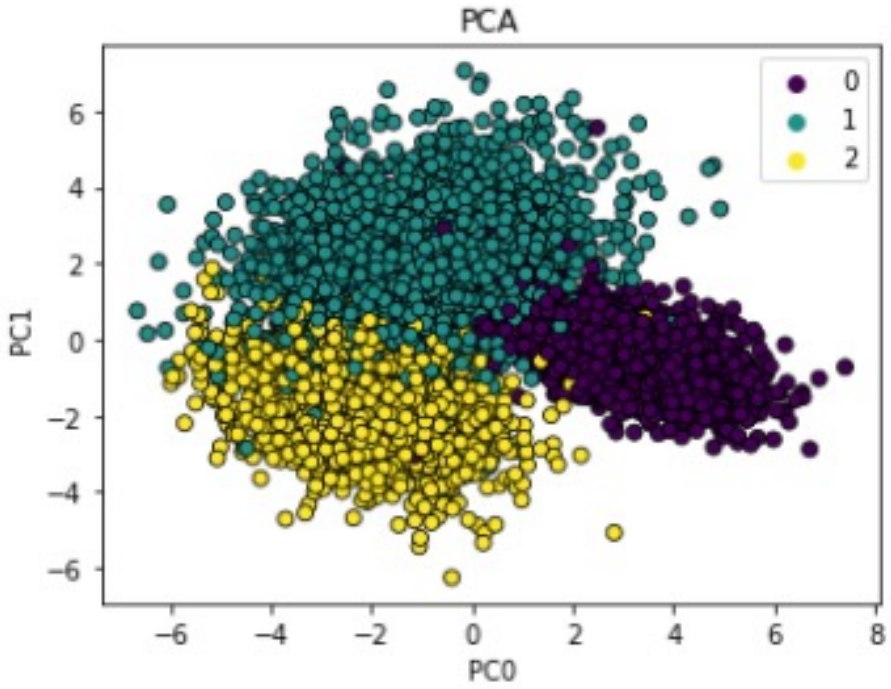
nn.unflatten(7×7×512)

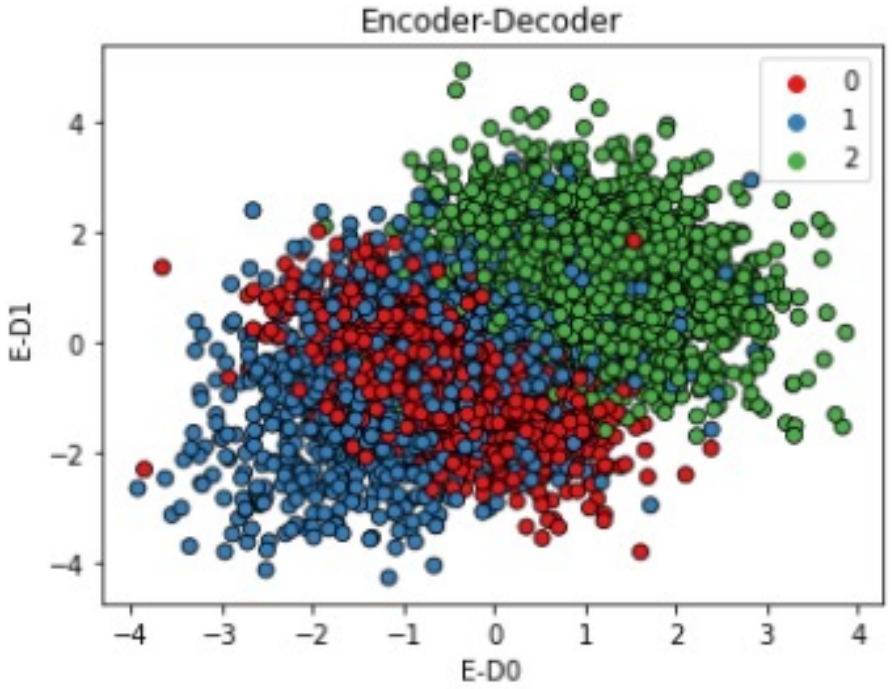
x.view(-1, 7, 7, 512)


2^n
x_1
x_2
x_3
x_{25k}
2048
1024
512
128
64
2

AutoEncoders for Dimensionality Reduction

- The encoder output can be used for dimensionality reduction

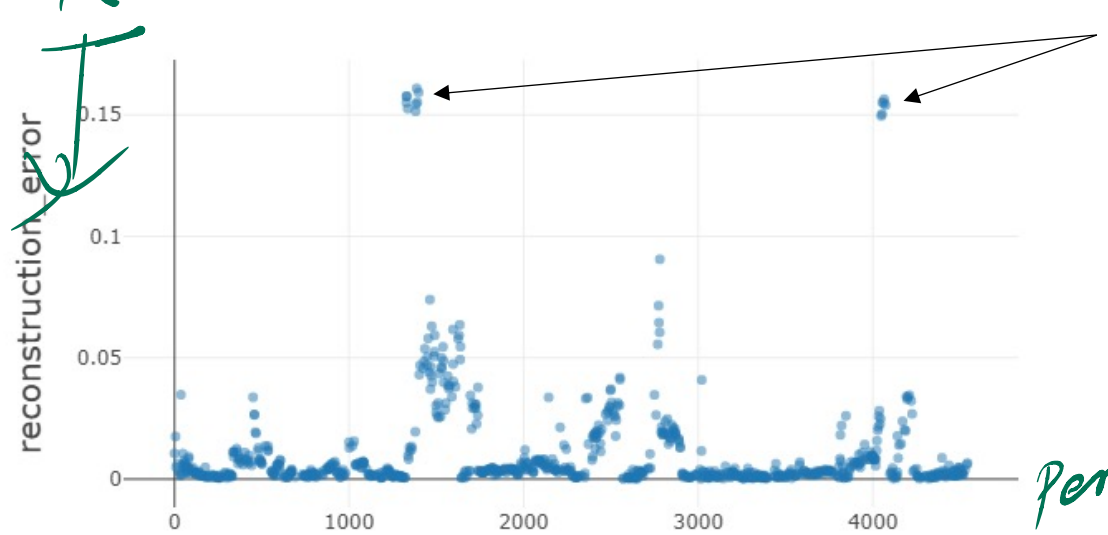




https://towardsdatascience.com/dimensionality-reduction-with-autoencoders-versus-pca-f47666f80743

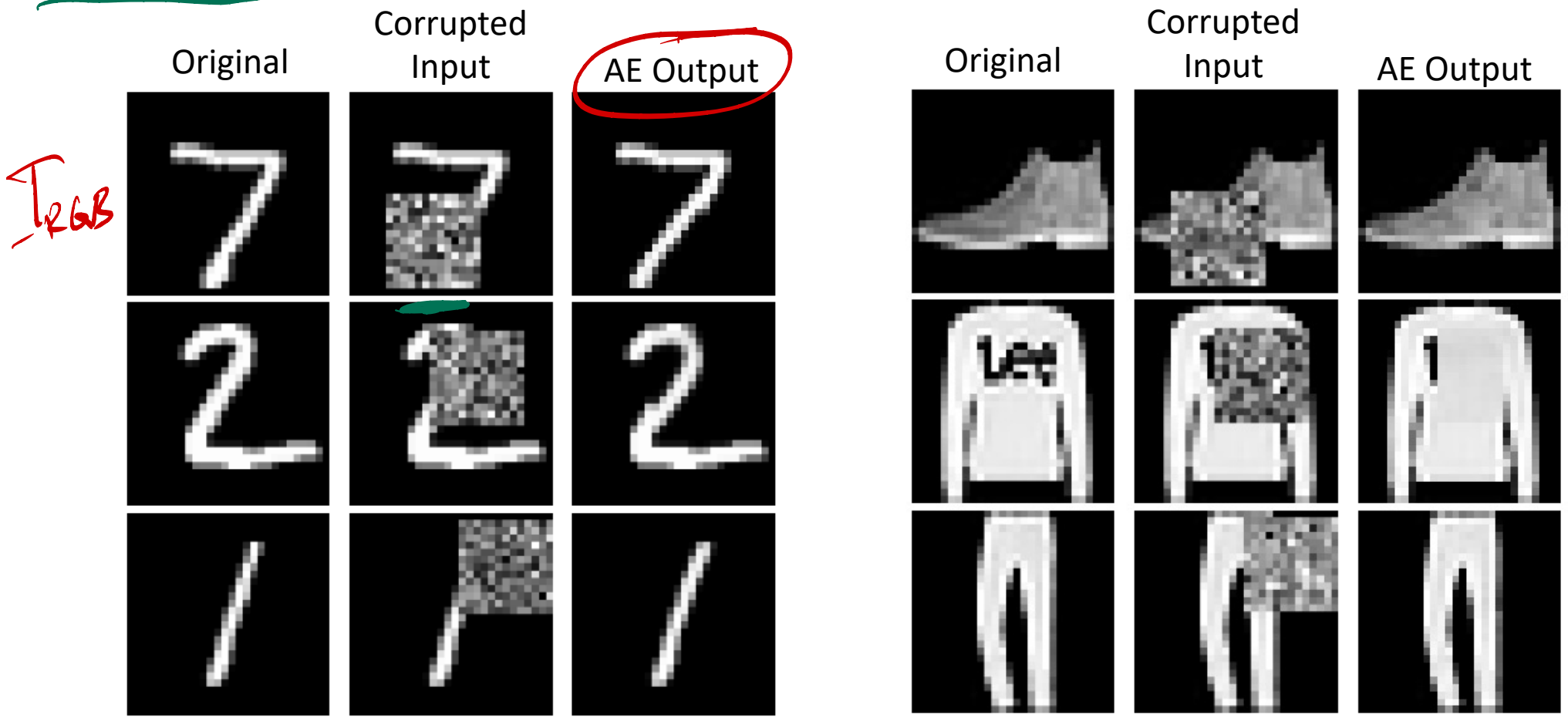
AutoEncoders for Outlier Detection

- The reconstruction error can be used to detect outliers. Out of distribution samples will have high Reconstruction error.



https://towardsdatascience.com/dimensionality-reduction-with-autoencoders-versus-pca-f47666f80743

AutoEncoders for Image Completion



Smieja et. al. Can auto-encoders help with filling missing data?

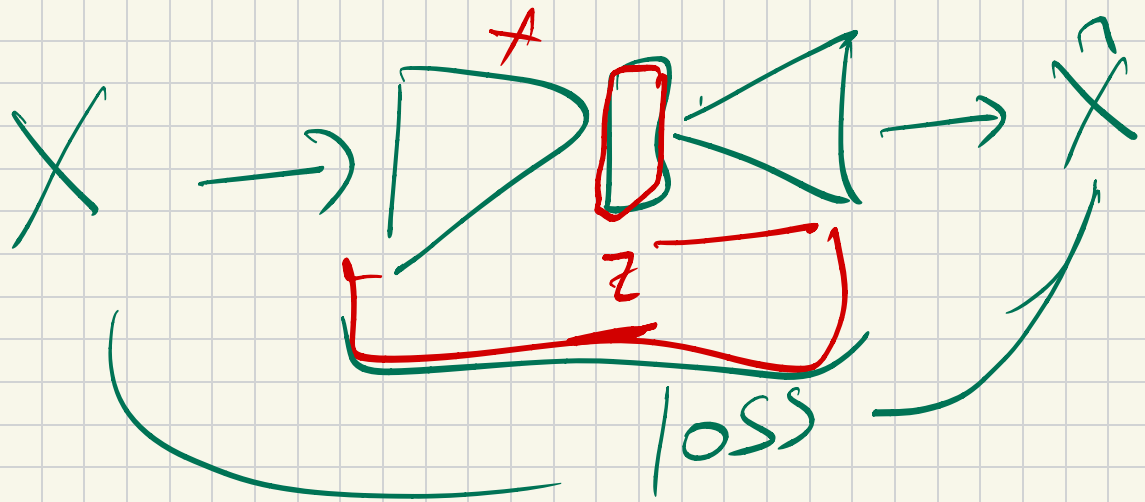
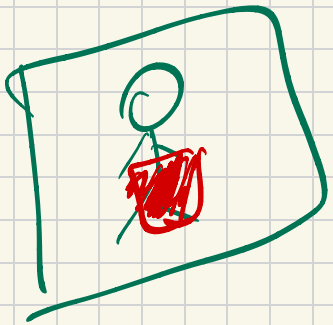
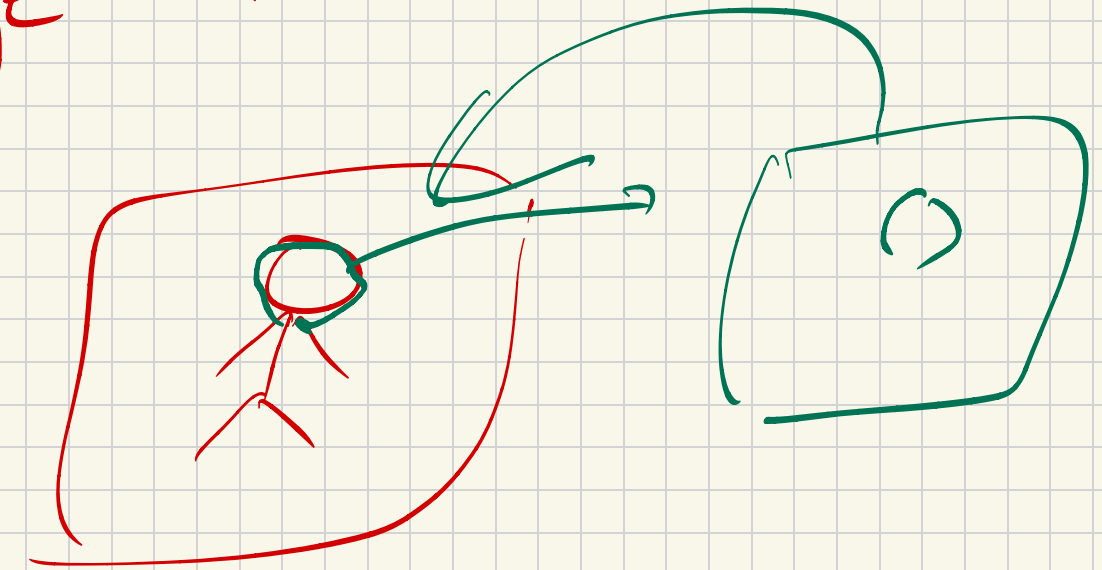


Image Generation





ML Algorithms Perspectives:

- We can look into ML algorithms from two perspective:
  - **Loss Minimization** Problem (like what we did).
  - **Probability Maximization** Problem (using Maximum Likelihood Estimation).
- Both should result in the same solution.

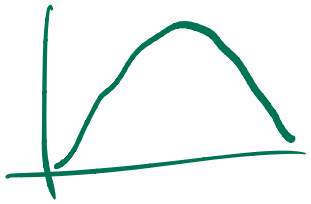
Probalistic Interpretration of Linear Regression and MLE

- We can also look at the probalistic Interpretation of Linear Regression.
- Keeping everything else same as the previous formulation

$$y_i = \boldsymbol{x}_i^T \boldsymbol{\theta} + \epsilon_i$$

- Now assume that $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, then $y_i \sim \mathcal{N}(\boldsymbol{x}_i^T \boldsymbol{\theta}, \sigma^2)$
- We can write the conditional distribution as :

$$\mathbb{P}(y_i | \boldsymbol{x}_i) \sim \mathcal{N}(0, \sigma^2)$$



Probalistic Interpretation of LR

- Let's assume that all data point in the dataset are i.i.d. (independent identically distributed). Then we have:

$$\mathbb{P}(\mathcal{D}) = \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i, y_i)$$

- Using Bayes Theorem we can write:

$$\prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i, y_i) = \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i)\mathbb{P}(y_i|\boldsymbol{x}_i)$$

$$P(\text{the teacher drinks water}) =$$

$$P(\text{the})\,P(\text{teacher}\mid\text{the})\,P(\text{drinks}\mid\text{the teacher})\cdot$$

$$P(\text{water}\mid\text{the teacher drinks})\cdot$$

Maximum Likelihoold Estimator

- In simple words, given the Dataset we want to find the values of the unknown parameters which maximize the probability of the Dataset.
- Using the definition of the conditional distrubtion we have

$$\mathbb{P}(y_i|\boldsymbol{x}_i) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})\right)$$

- Using the definition we get

$$\prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i, y_i) = \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i) \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})\right)$$

Maximum Likelihood Estimator

- Let's try to maximaize:

$$\prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i, y_i) = \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i) \prod_{i=1}^{N} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})\right)$$

- Note that

$$\arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{x}_i, y_i) = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{N} \exp\left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})^2\right)$$

Maximum Likelihood Estimator

- Furthermore, since the right hand side of the above equation is monotonice in \theta the arg max will not change if we take log of the expression

$$\arg\max_{\theta} \prod_{i=1}^{N} \exp\left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})^2\right) = \arg\max_{\theta} \sum_{i=1}^{N} \left(-(y_i - \boldsymbol{x}_i^T\boldsymbol{\theta})^2\right)$$

- Notice that the right hand side is minimising the MSE.
- Hence solution of minimizing the MSE is equivalend to Maximum Likelihood Estimator for linear regression

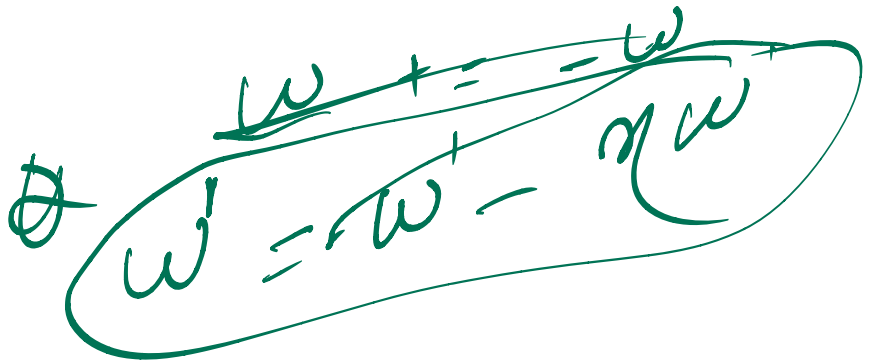
A Slight Detour: A Look at Optimization Tools

Introduction

- **Optimizers** are algorithms that adjust the weights of the model to minimize the loss function during training.
- They are one of the main components of Deep Learning models.
- We will go through several types of optimizers in this section.

Direction of maximum increase and decrease for a function

- Gradient direction is the direction of maximum increase for a function
- Negative gradient is the direction of maximum decrease for a function



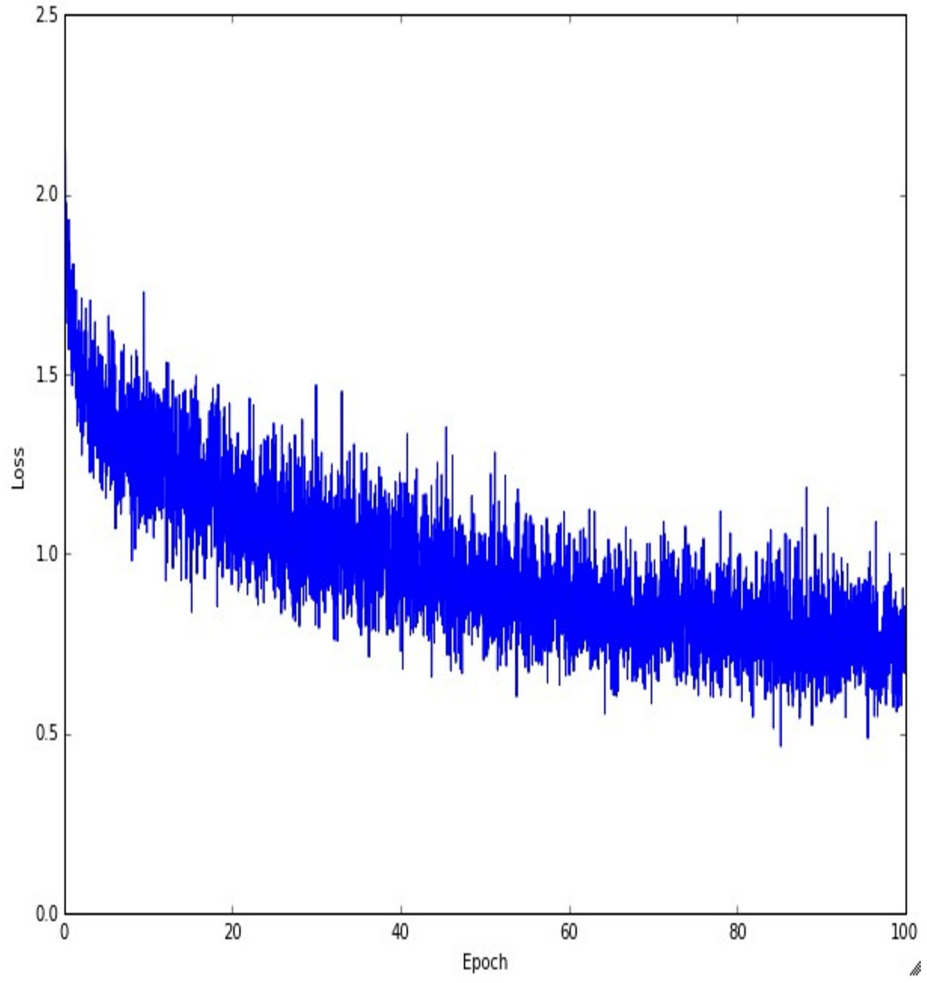
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- **Mini batch Gradient Descent**
- only use a small portion of the training set to compute the gradient.

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

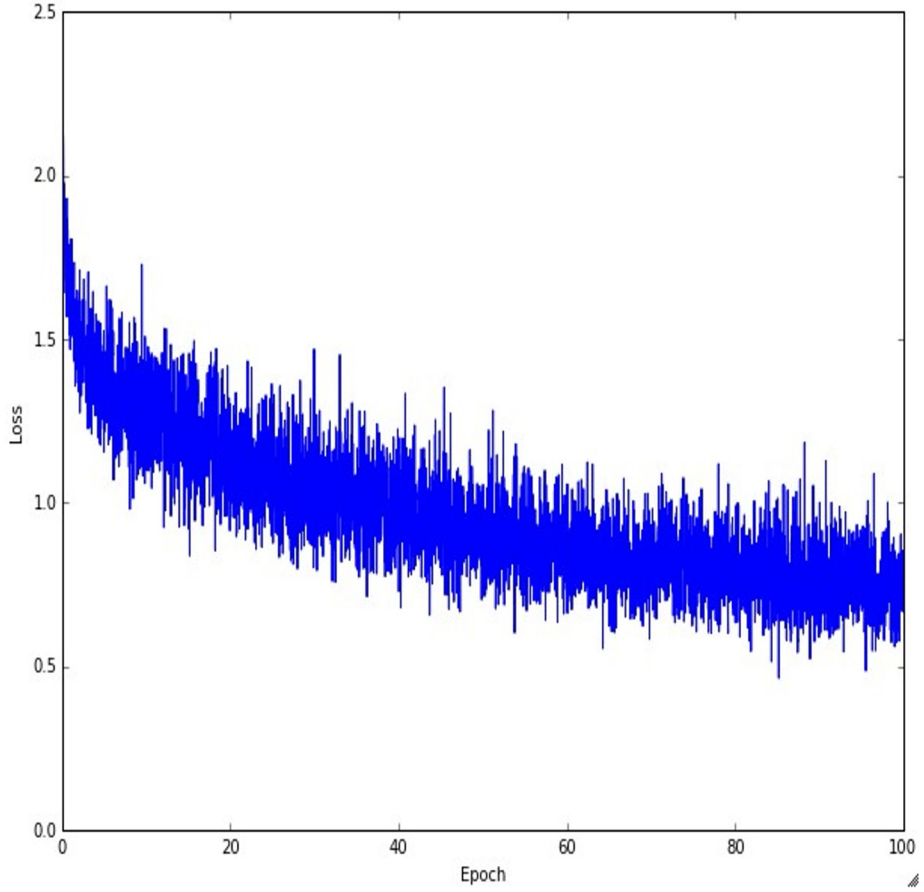
Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

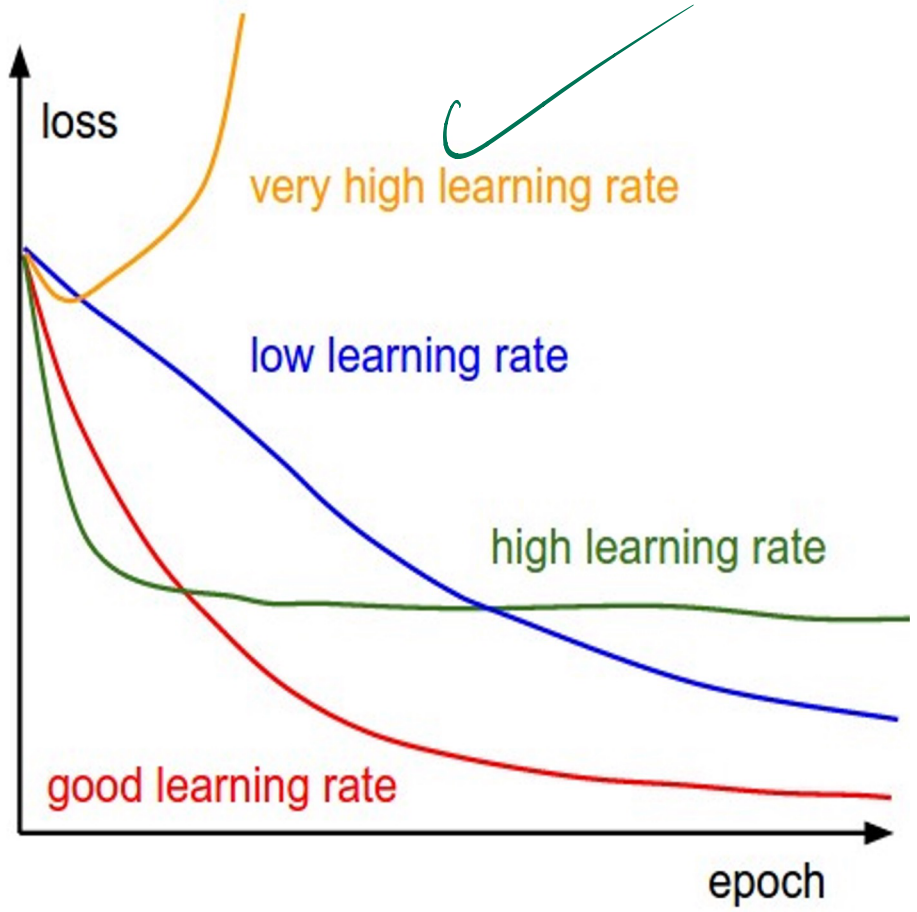
2^n



Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)





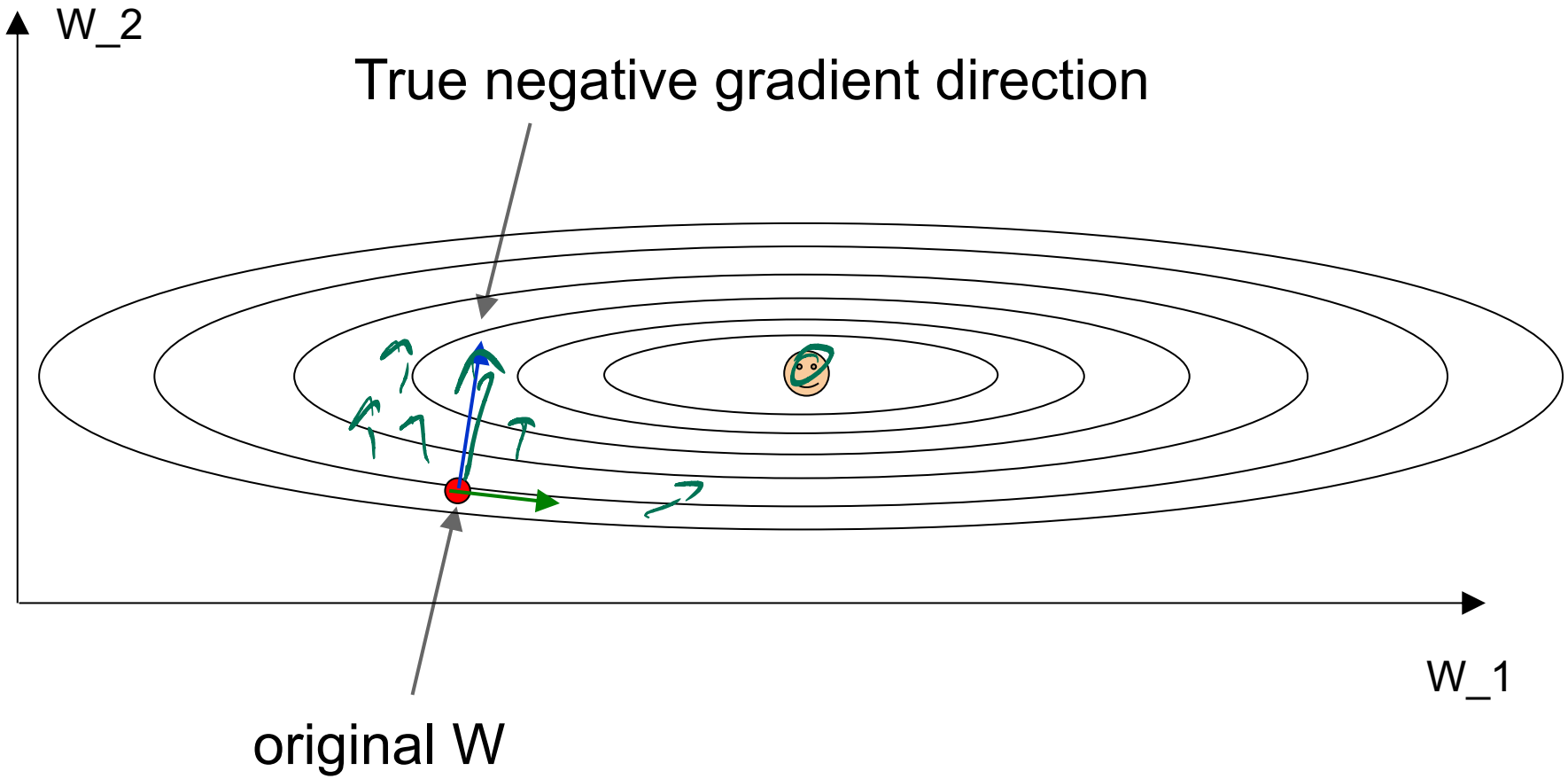
Slides based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

- only use a small portion of the training set to compute the gradient.

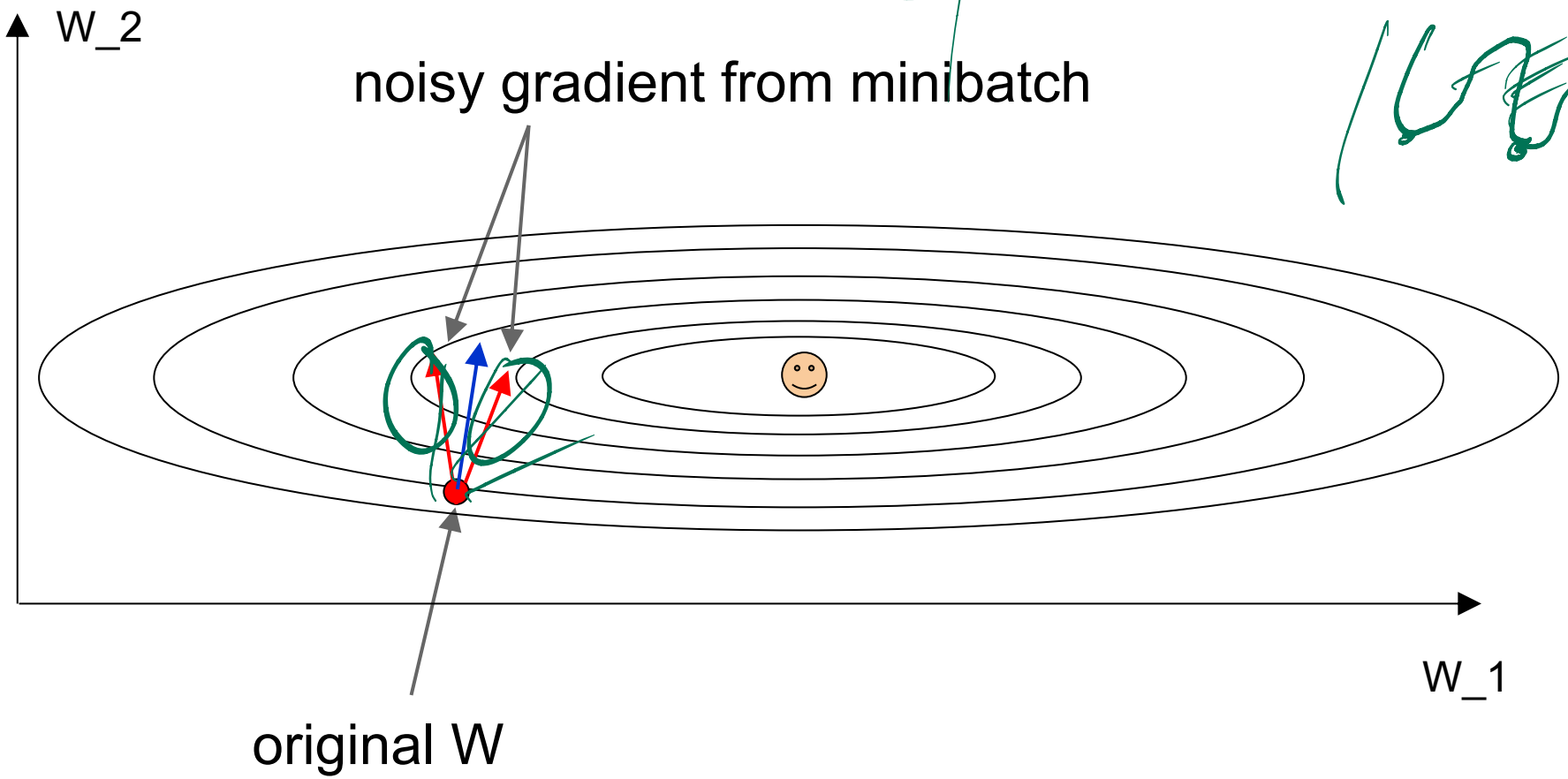
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

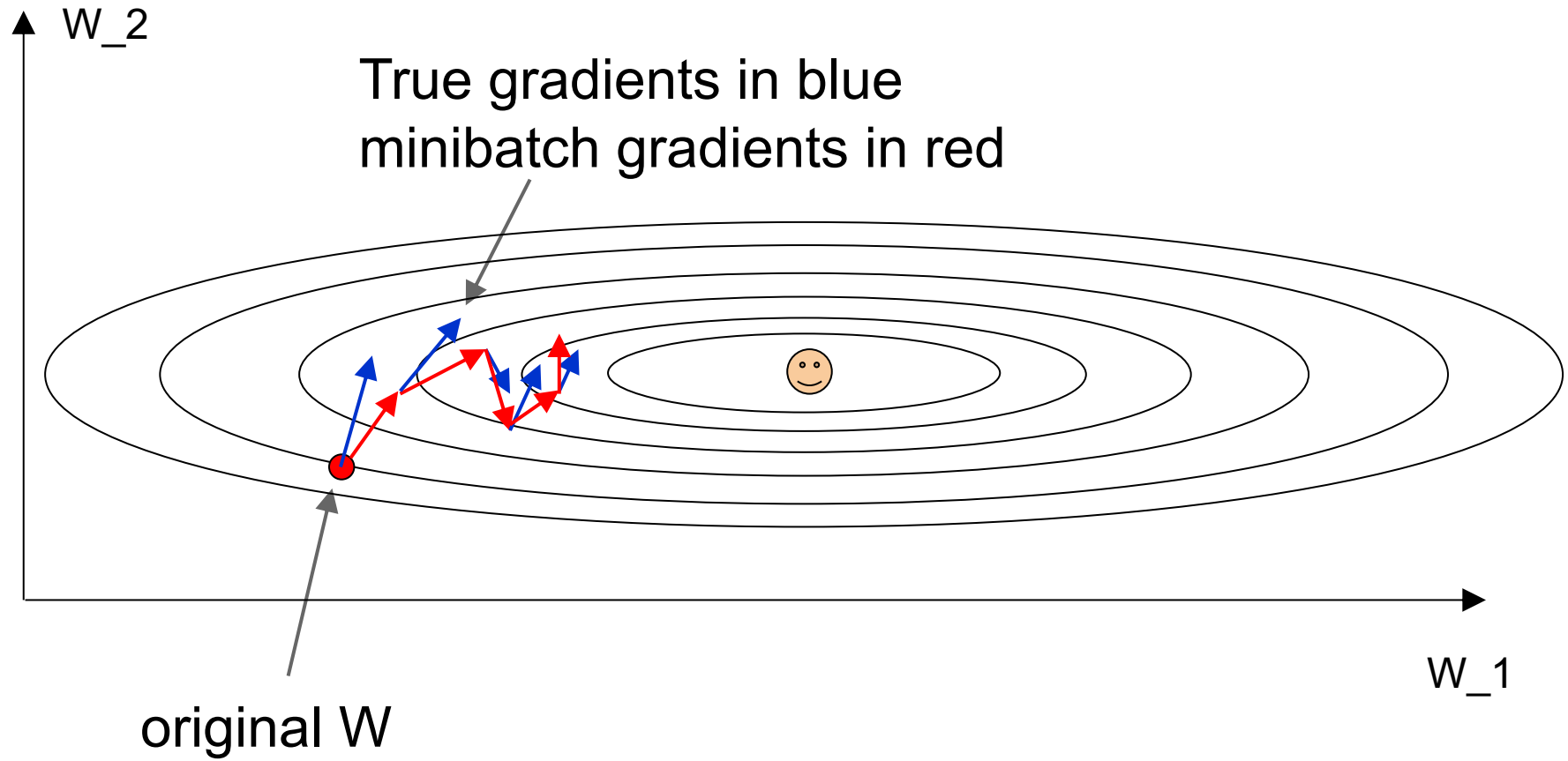
we will look at more fancy update formulas (momentum, Adagrad, RMSProp, Adam, …)



Slide based on CS294-129 by John Canny

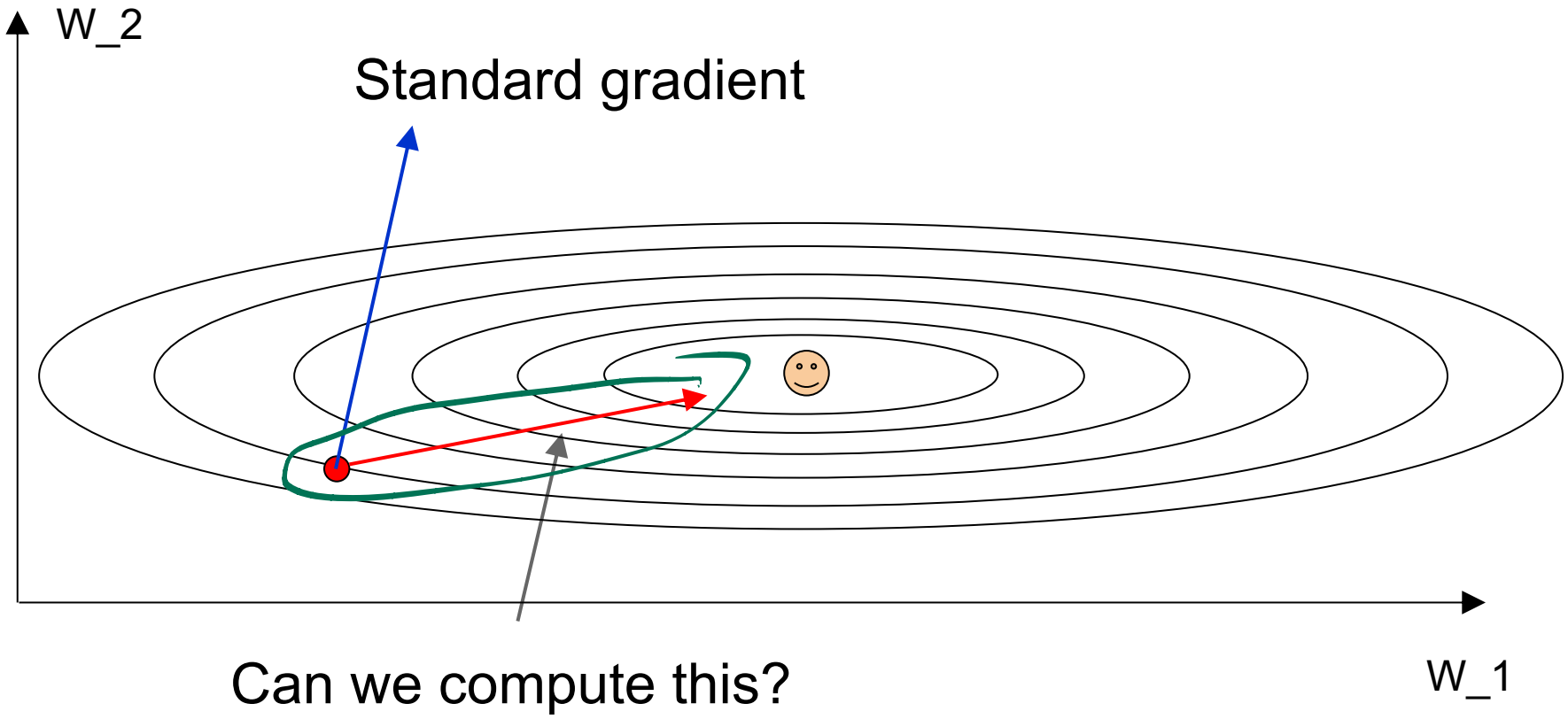


Slide based on CS294-129 by John Canny

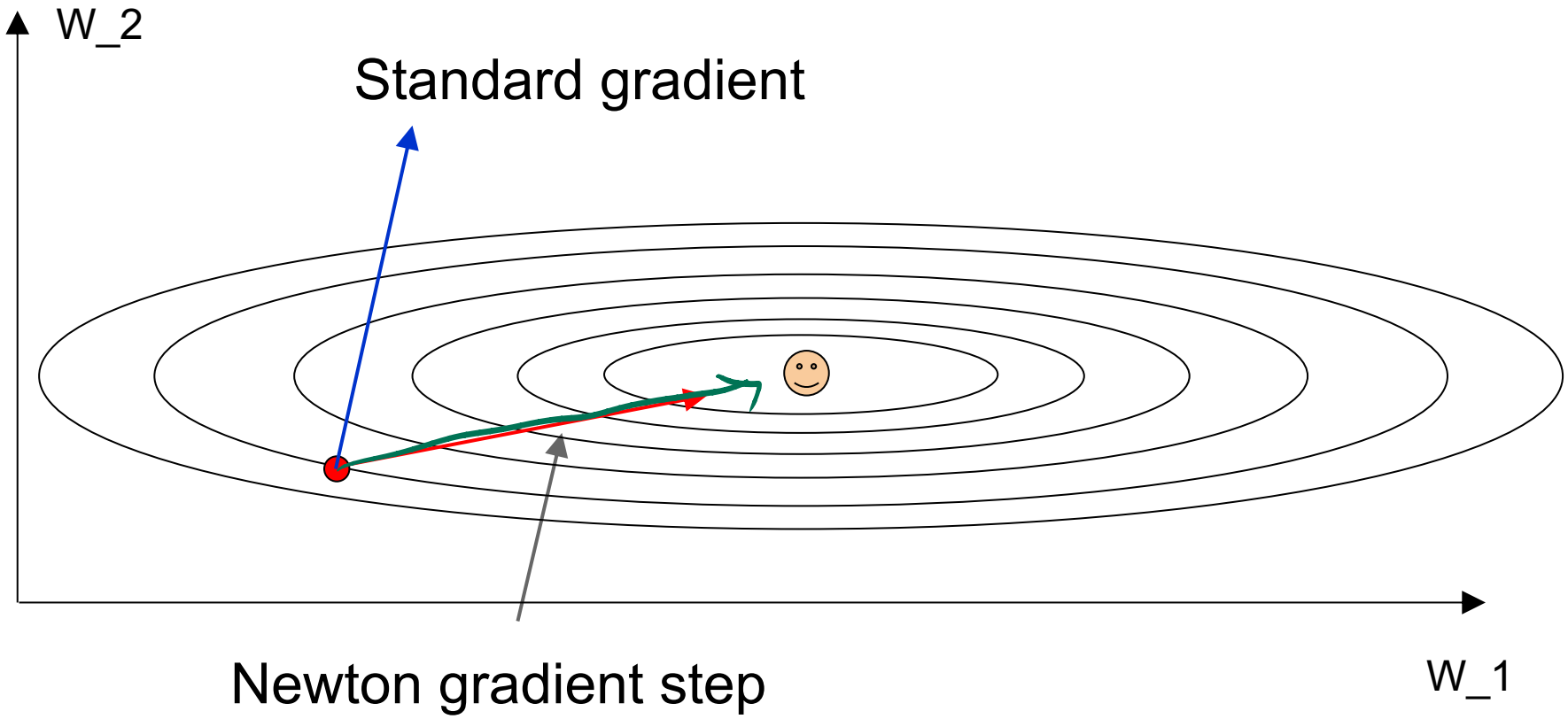


Gradients are noisy but still make good progress on average

Slide based on CS294-129 by John Canny



Slide based on CS294-129 by John Canny



Slide based on CS294-129 by John Canny

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1}\nabla f(x_n)$$

Where H_f is the Hessian matrix of second derivatives of f.

Converges very fast, but rarely used in DL. Why?

Slide based on CS294-129 by John Canny

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Slide based on CS294-129 by John Canny

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Too unstable: it involves a high-dimensional matrix inverse, which has poor numerical stability. The Hessian may even be singular.

Slide based on CS294-129 by John Canny

Momentum Optimization

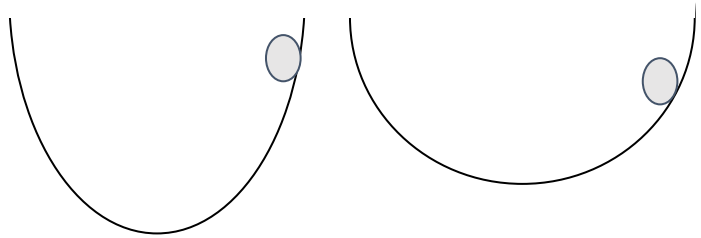
```
# Gradient descent update
x += - learning_rate * dx
```



```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99) for adaptation

```
# Gradient descent update
x += - learning_rate * dx
```



```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

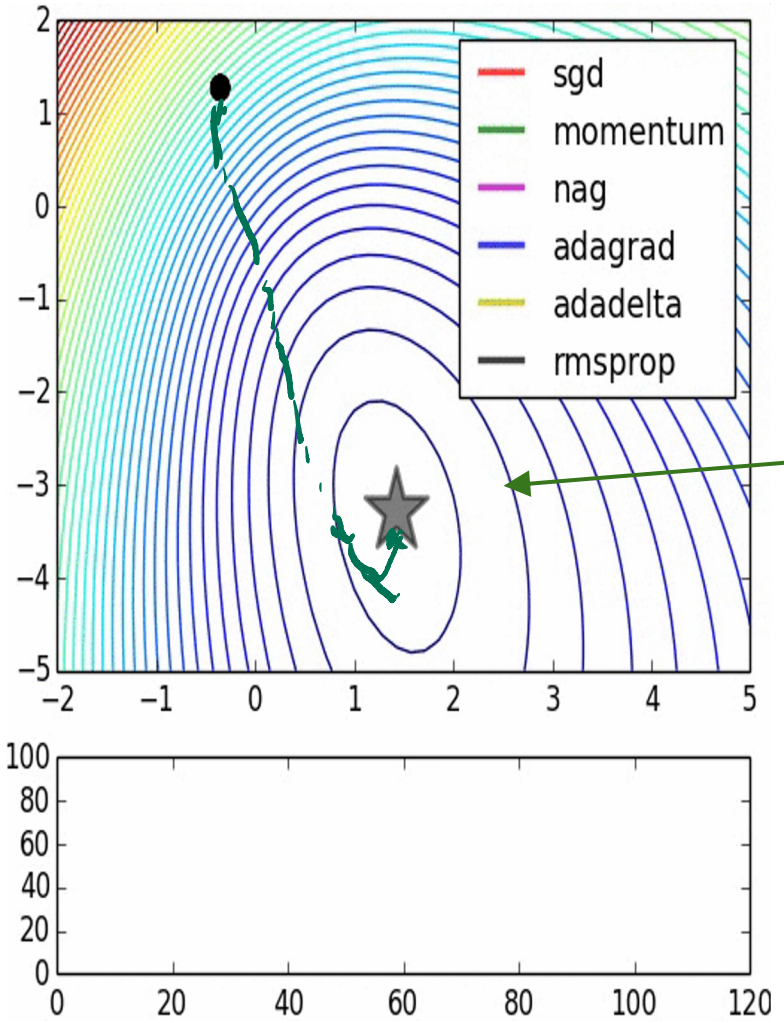
- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

Smoother trajectories in optimization, avoiding zig-zagging behavior.
Faster convergence towards the global minimum.

SGD

vs

Momentum



notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

[Duchi et al., 2011]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

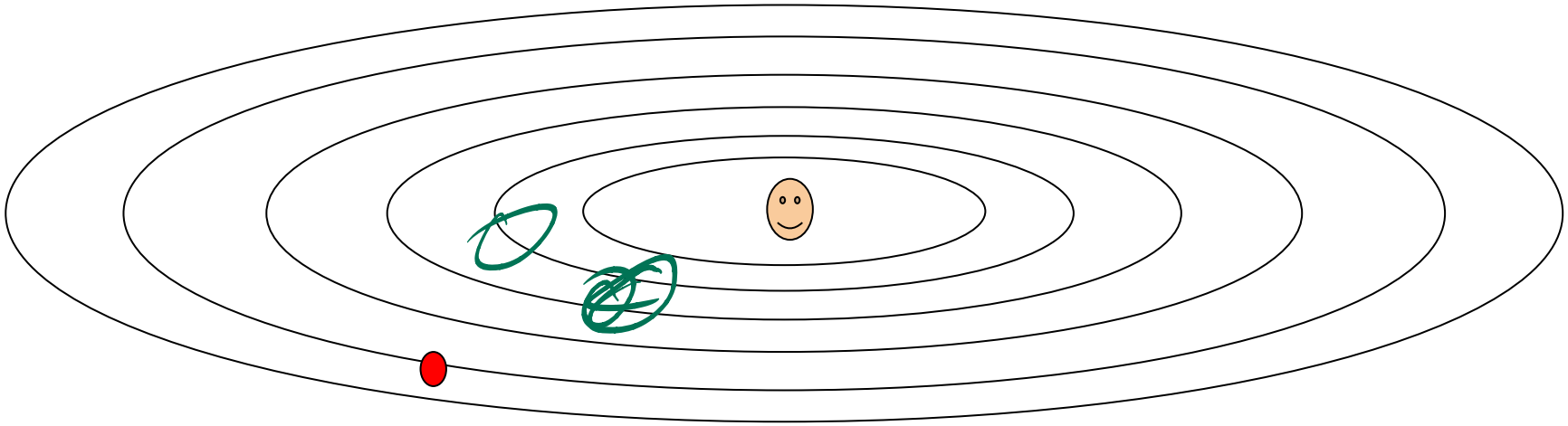
gradient clipping

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

- The element-wise scaling ensures that parameters corresponding to dimensions with larger cumulative gradients (cache) are updated less aggressively.
- Conversely, parameters with smaller cumulative gradients are updated more, balancing their influence.

making larger adjustments in directions where gradients are smaller (flatter regions) and smaller adjustments in steeper directions.

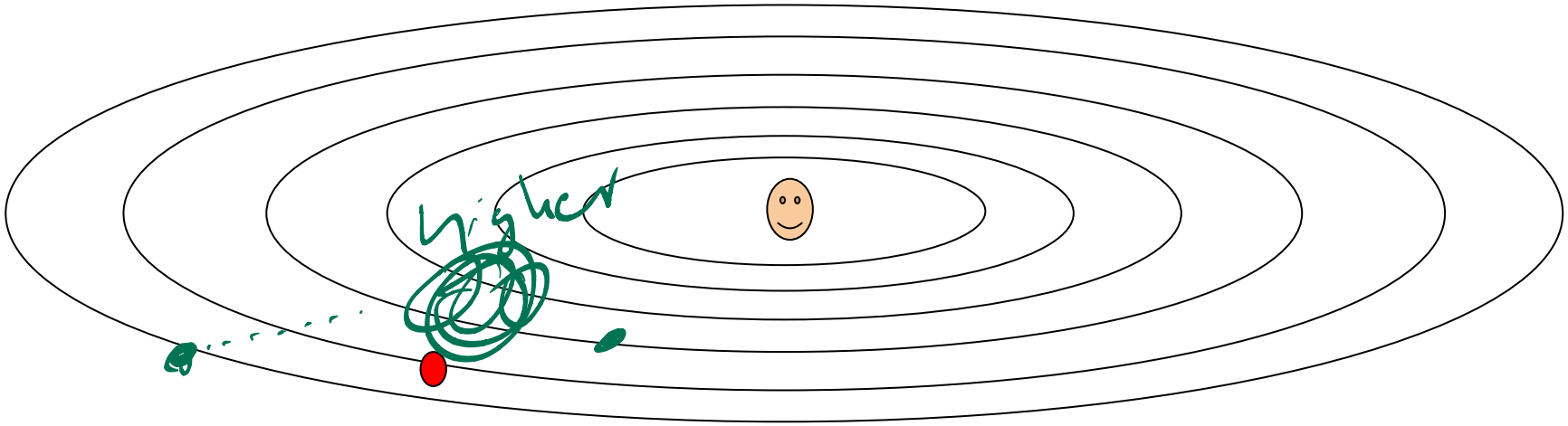
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q: What happens with AdaGrad?

The optimizer takes smaller steps vertically (steep direction) and larger steps horizontally (flat direction).

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q2: What happens to the step size over long time?

Extremely small steps as the optimizer nears convergence, effectively "stalling" progress.

RMSProp

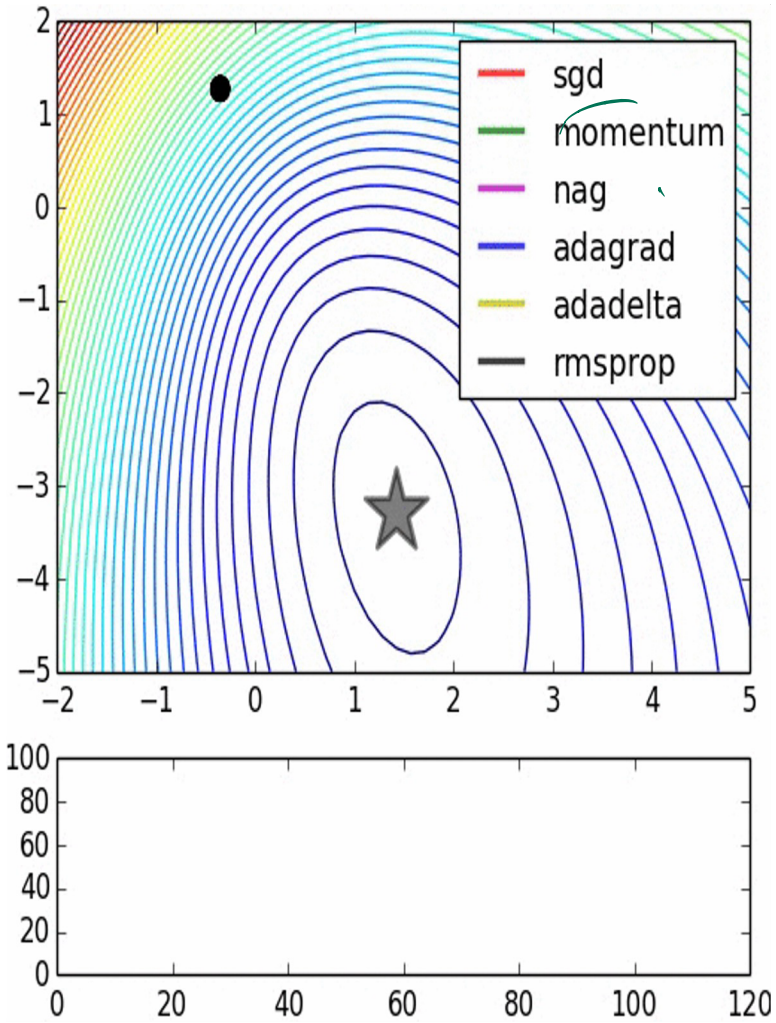
[Tieleman and Hinton, 2012]

Exponential decay factor for cache, which prevents it from growing indefinitely.

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



adagrad
rmsprop

[Kingma and Ba, 2014]

Adam optimizer

smooths the gradients over time by keeping a moving average of the gradient.

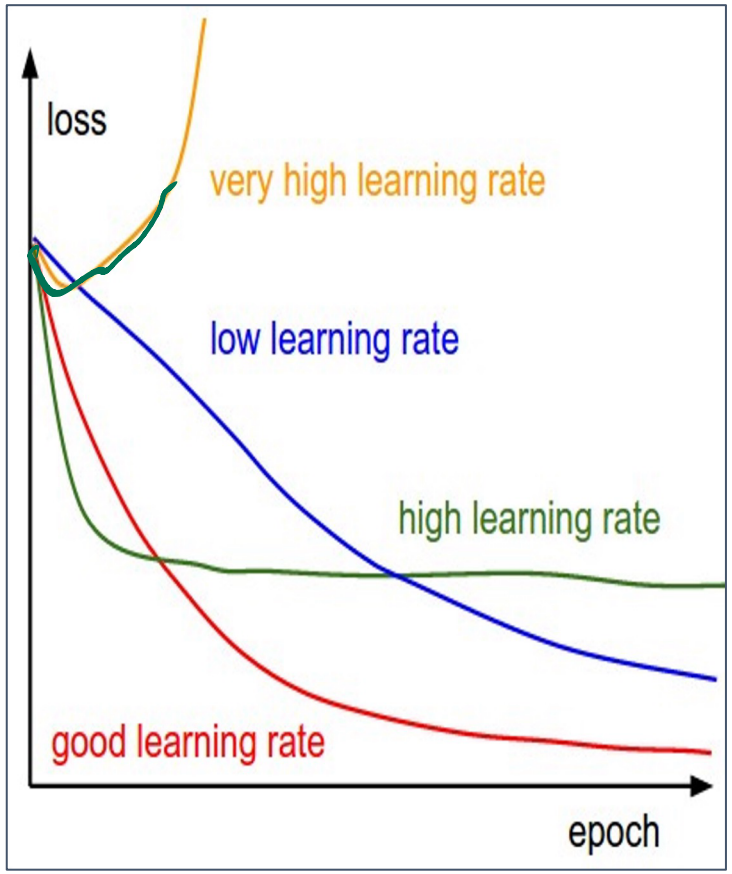
```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

normalizing gradients by scaling them with the square root of their historical squared values.

Looks a bit like RMSProp with momentum



Q: Which one of these learning rates is best to use?

model.train()

for img, gt i - - - :

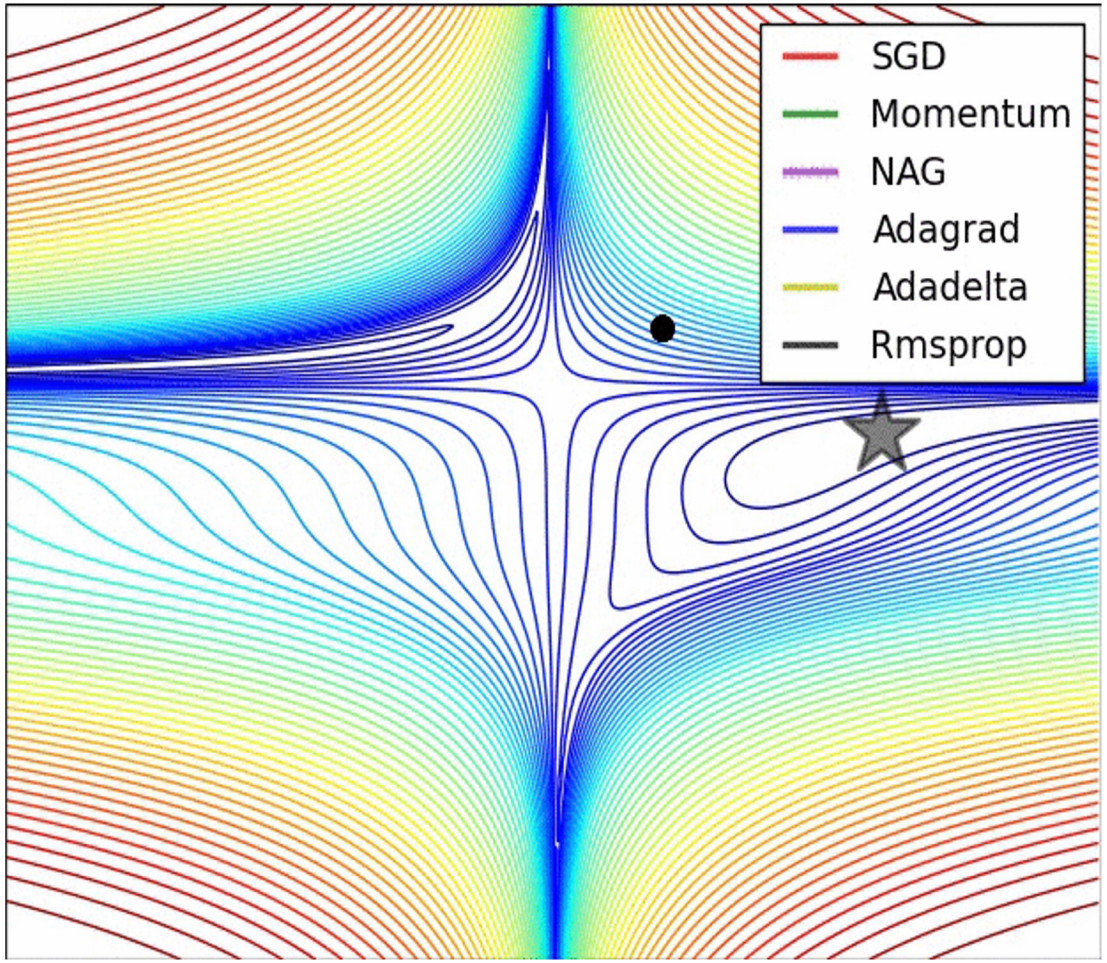
out = m.(img)

loss = CE(out, gt)

-> to, t = ip loss = :

If epoch - - -

adjust_ - - (   )



(image credits to Alec Radford)

Summary