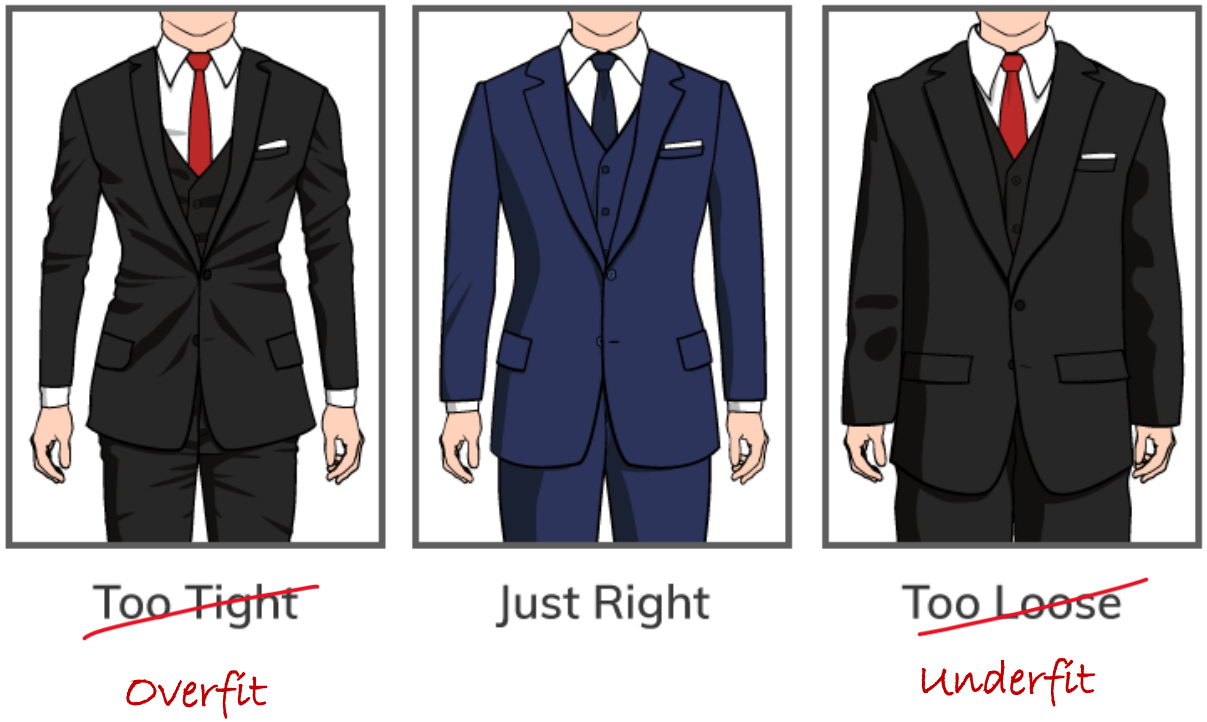
Optimizers and regularization

Image by Agnes Monkelbaan published under CC 4.0

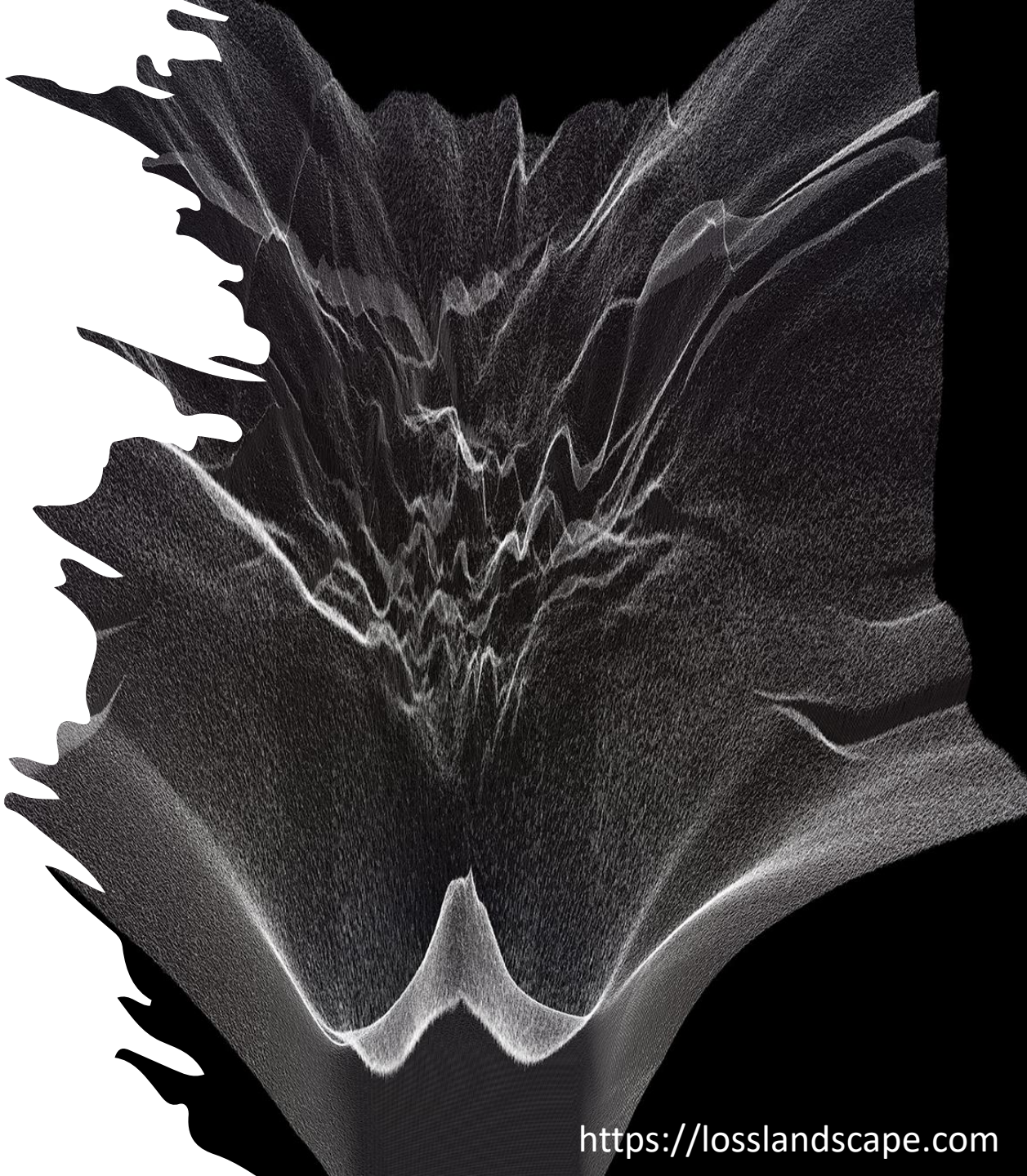
Today's topics

- Optimizers
  - SDG
  - Adam
- Regularization
  - L2
  - Dropout
  - Early stopping
  - Batchnorm
  - Data augmentation



Optimizers

Optimization is the process of finding the set of parameters that minimize the loss function.



Optimizer strategy 1: Random search

- We could generate random weights and keep track of what works
- Few iterations might give results better than the baseline, <u>but we can do much better</u>

```
for num in range(1000):
  W = np.random.randn(10, 3073) * 0.0001
  loss = L(X_train, Y_train, W)
  if loss < bestloss:
    bestloss = loss
    bestW = W
```

Optimizer strategy 2: Following the Gradient

- *Gradient Descent*
  - Repeatedly evaluate the gradient of the loss function of the full training set and step in the direction that minimizes the loss the most.

```
For every Epoch:
   θ := θ - α ∇θJ(θ)
```

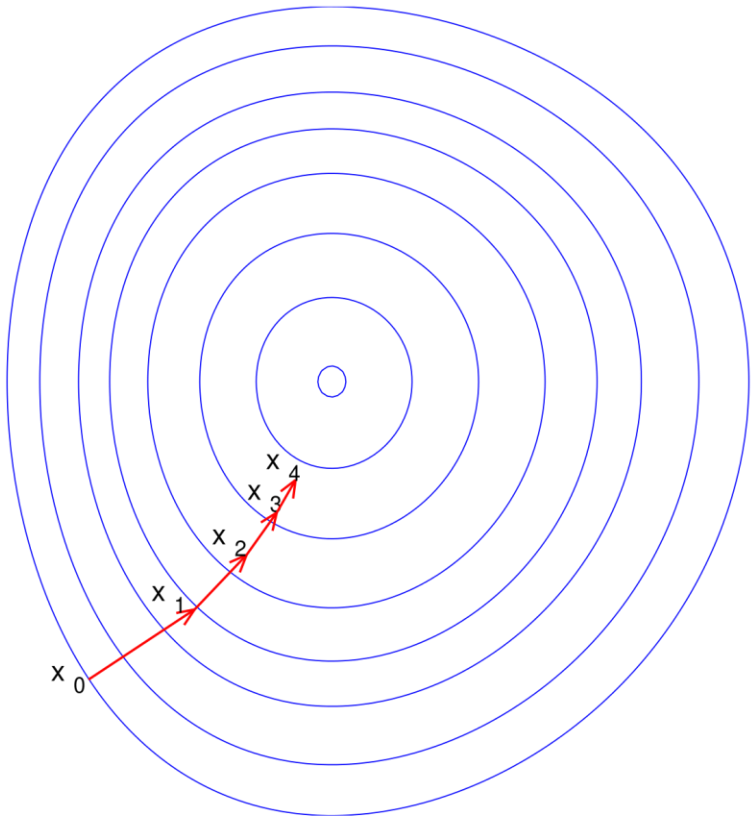
Cost function

Updated parameters

Existing parameters

Learning rate (step size)

gradient



θ=(w)

Optimizer strategy 2: Following the Gradient

- *Gradient Descent*
  - Repeatedly evaluate the gradient of the loss function of the full training set and step in the direction that minimizes the loss the most.

```
For every Epoch:
    θ := θ - α ∇θJ(θ)
```

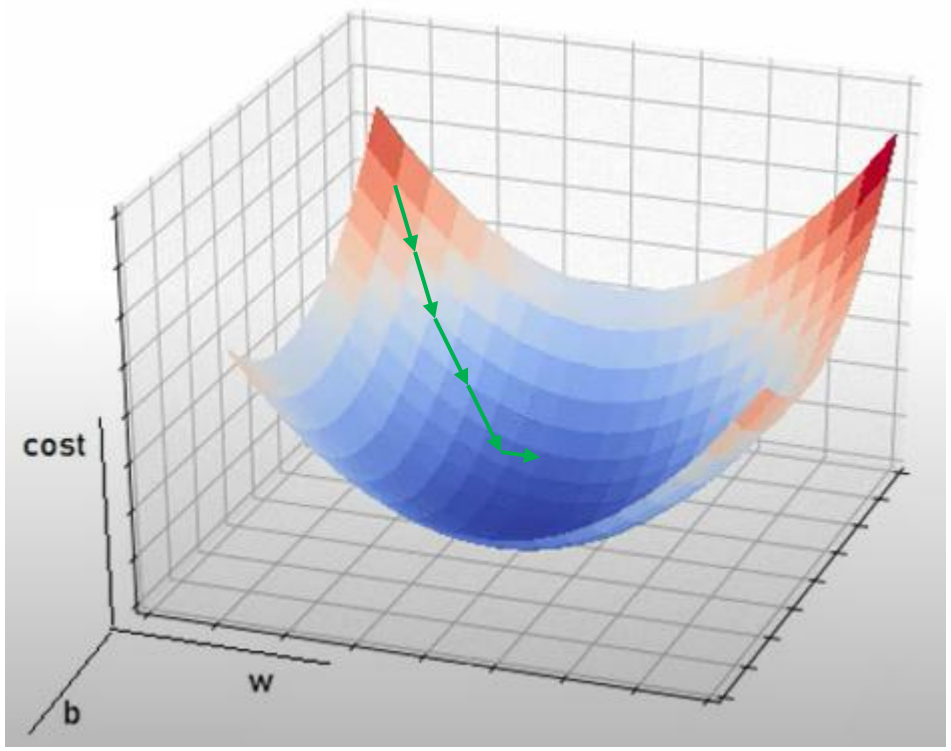
Cost function

Updated parameters

Existing parameters

Learning rate (step size)

gradient

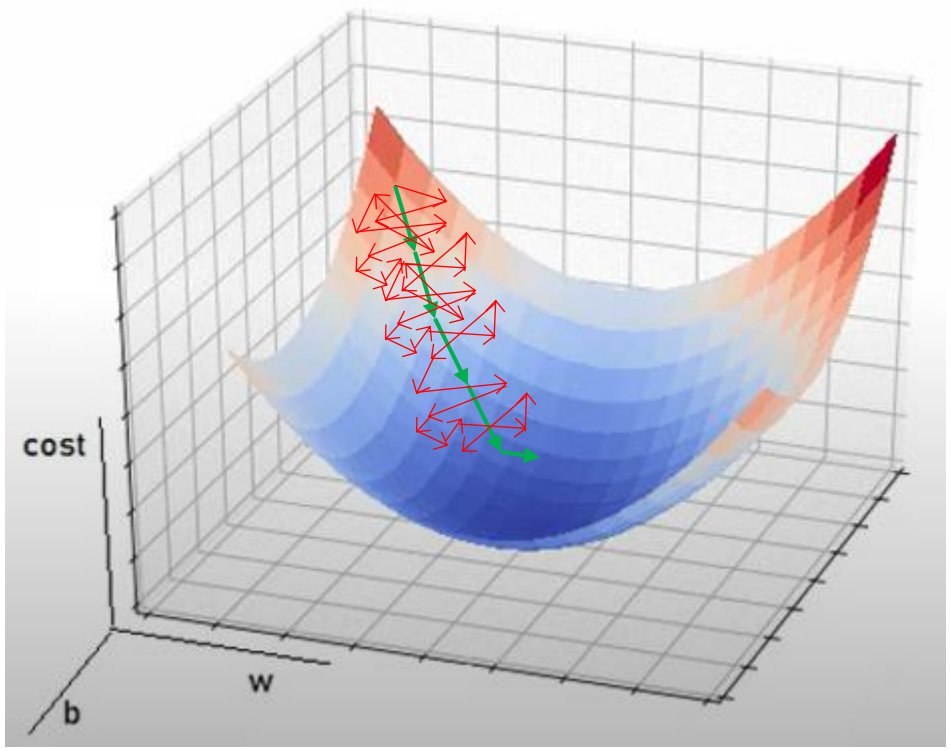


θ=(w, b)

Stochastic gradient descent

- In Gradient Descent we update the parameters based on the full dataset
- In Stochastic Gradient Descent we update the parameters based on a part of the dataset.
  - Leads to more frequent update
  - Leads to noisier update
    - May help us escape local minima

```
For every Epoch:
    For every sample:
        θ := θ - α ∇θJ(θ)
```

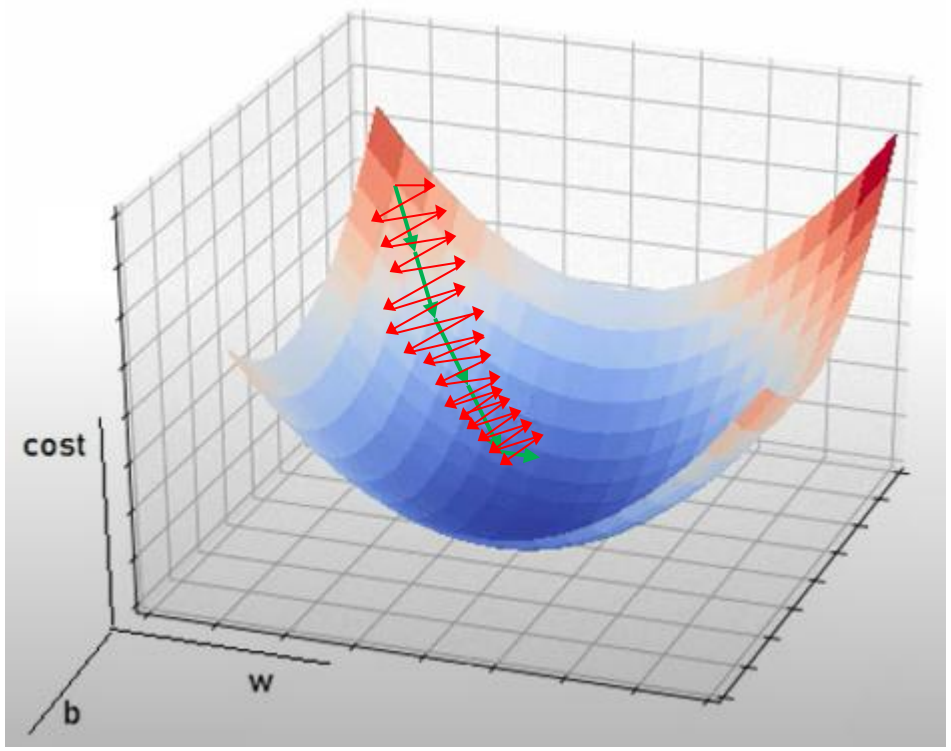


$\theta=(w, b)$

Compromise: Minibatch gradient descent

- As a compromise between updating the parameters after the full training set and updating the parameters after every sample, we can update after a minibatch of samples
- Quick update and less noisy

```
For every Epoch:
    For every N samples:
        θ := θ - α ∇θJ(θ)
```



θ=(w, b)

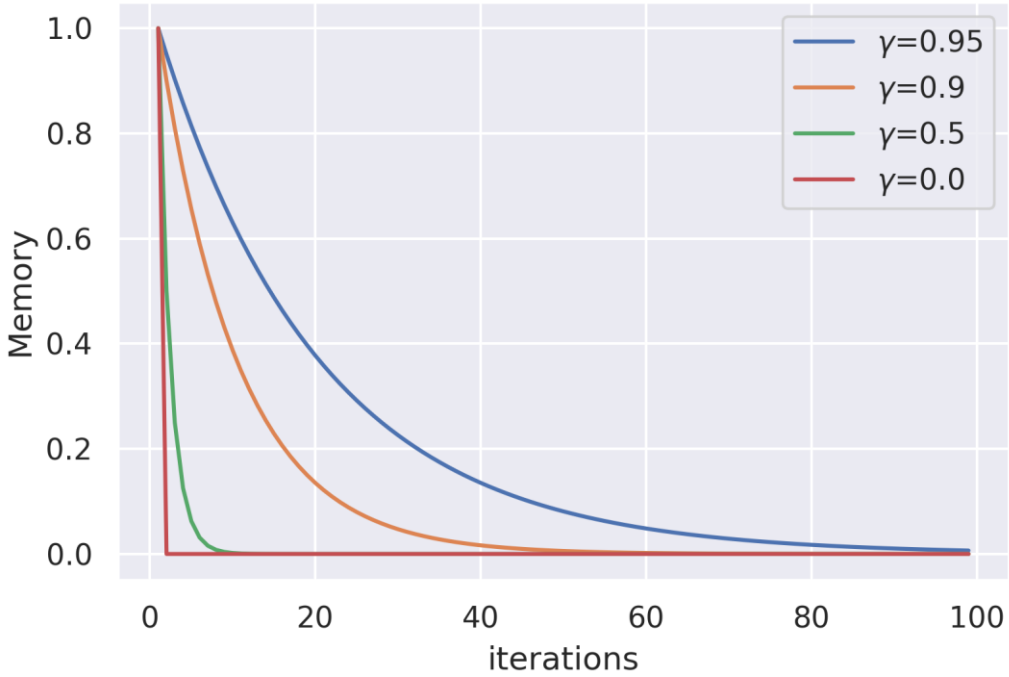
Momentum

- Momentum is a way to reduce a noisy path by smoothing the parameter update
  - Exponentially weighted moving average:

$$g_t = \nabla_\theta J(\theta_{t-1})$$

$$v_t = \gamma v_{t-1} + (1-\gamma) g_t$$
$$\theta_t = \theta_{t-1} - \alpha v_t$$

$$v_0 = 0$$
$$v_1 = \gamma v_0 + (1-\gamma) g_1$$
$$v_2 = \gamma v_1 + (1-\gamma) g_2$$
$$v_3 = \gamma v_2 + (1-\gamma) g_3$$
$$\ldots$$



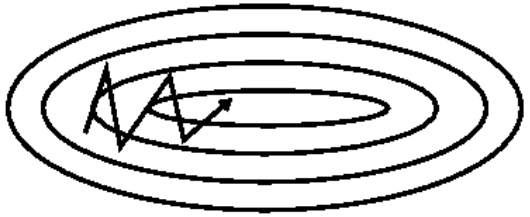

```
For every Epoch:
    For every (N) samples:
        vt = γ vt-1 + (1- γ) ∇θJ(θ)
        θt = θt-1 - α vt
```

γ = the momentum (0≤γ<1)
v = weighted moving average of previous gradients (velocity)
α = the learning rate

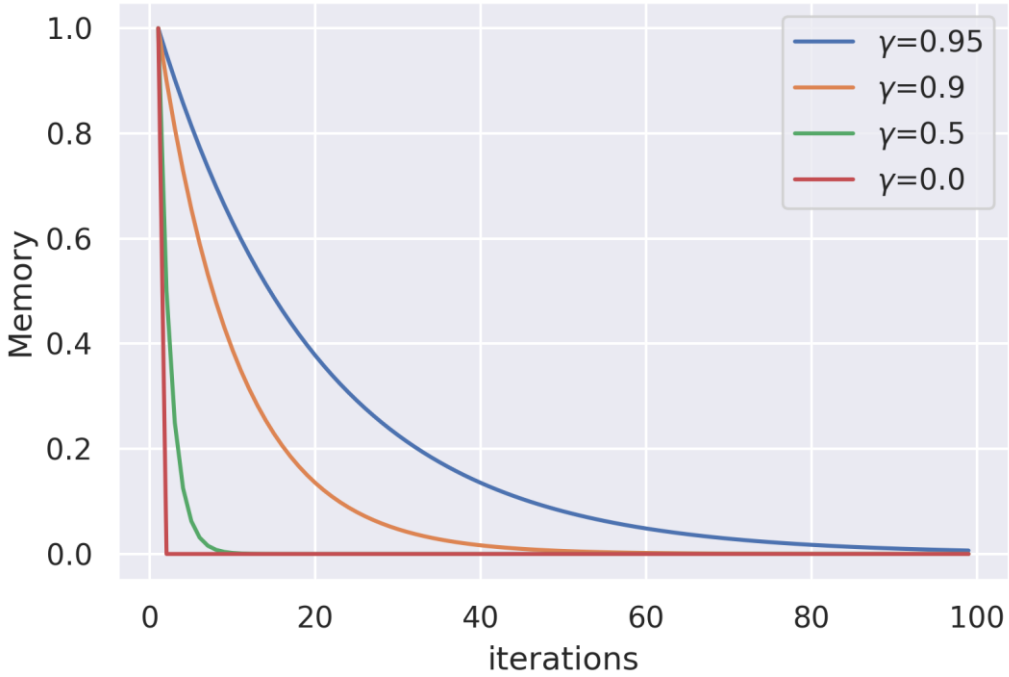
Momentum

- Momentum is a way to reduce a noisy path by smoothing the parameter update
- The updater will remember the direction from previous iterations and weight the current update direction with the previous ones
- By smoothing the weight updates across iterations, momentum tends to make deep learning with SGD both more stable and faster.

SDG without momentum



SDG with momentum




```
For every Epoch:
    For every (N) samples:
        v_t = γ v_{t-1} + (1- γ) ∇_θ J(θ)
        θ_t = θ_{t-1} - α v_t
```

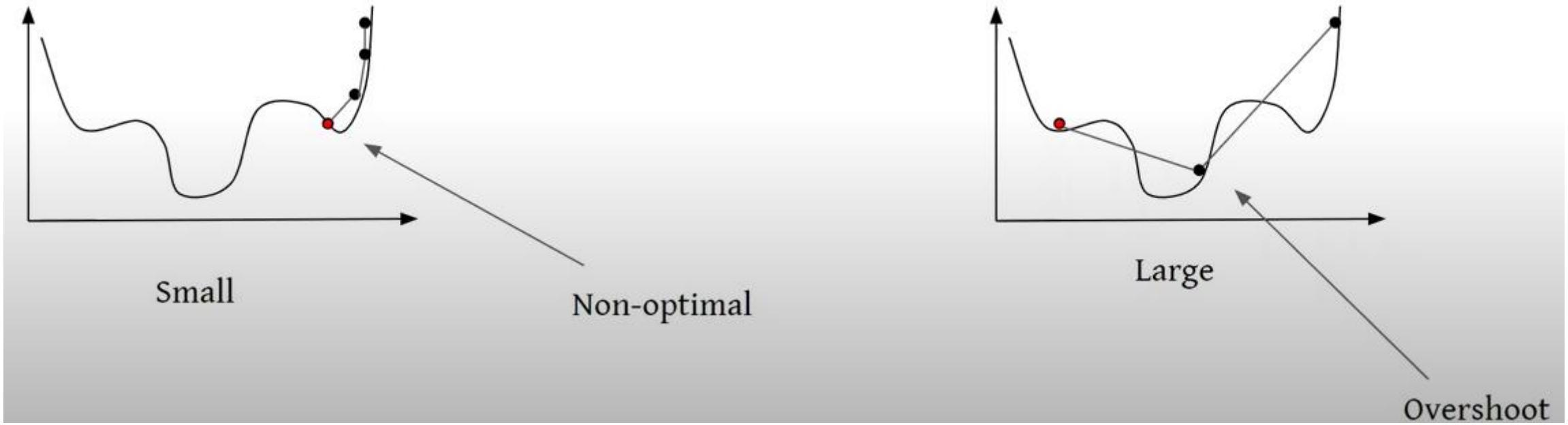
γ = the momentum (0≤γ<1)

v = weighted moving average of previous gradients (velocity)

α = the learning rate

Problem with vanilla gradient descent

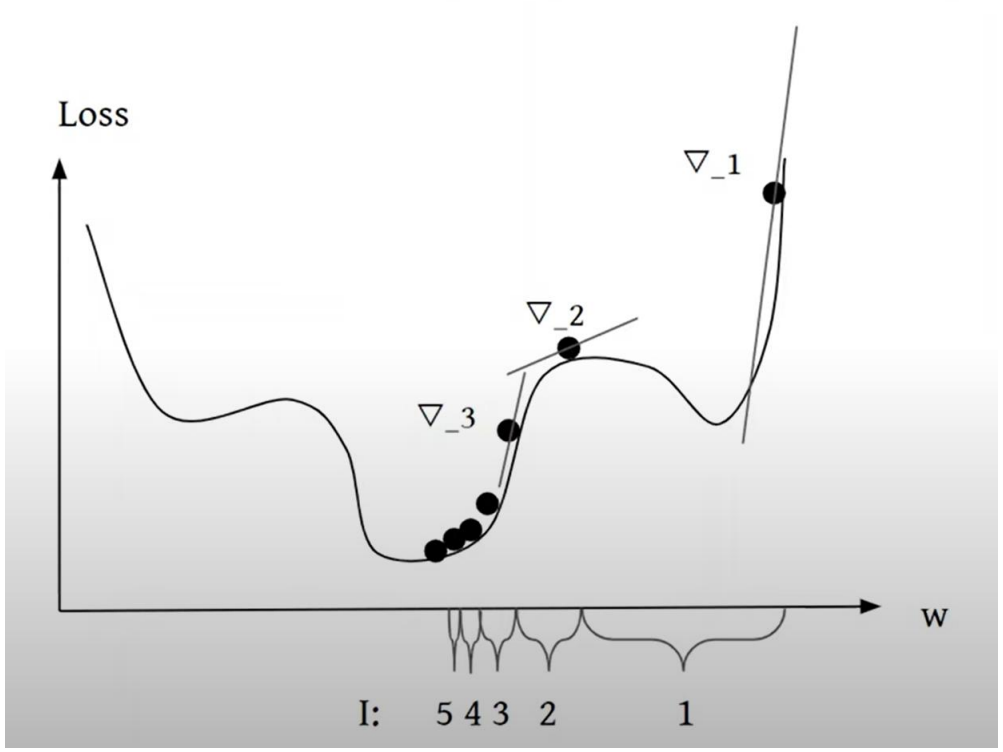
- Small learning rates are less likely to overshoot, but they are also more likely to converge at a non-optimal local minimum.
- Large learning rates converge faster, but are likely to overshoot



Adagrad (adaptive gradient algorithm)

Main idea: A modified stochastic gradient descent algorithm with per-parameter learning rate

- Change the learning rate α based on past gradients of a particular weight



$$I_1 = \frac{\alpha}{|g_1|}$$

$$I_2 = \frac{\alpha}{|\{g_1, g_2\}|}$$

$$I_3 = \frac{\alpha}{|\{g_1, g_2, g_3\}|}$$

....

where

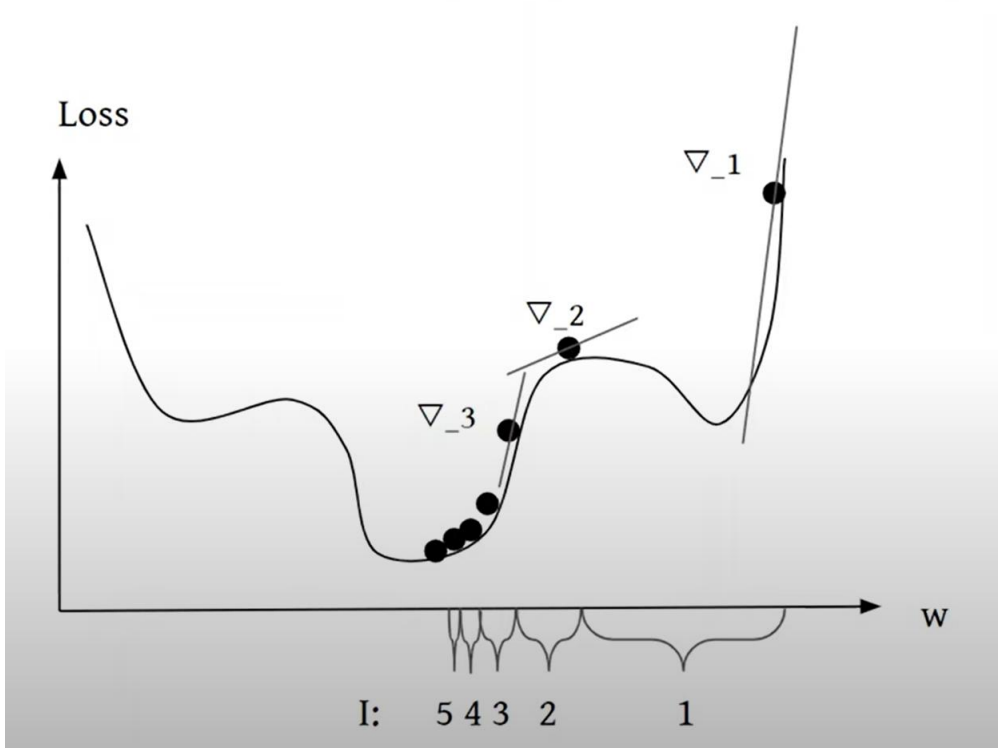
$$|\{g_1, g_2, \dots, g_t\}| = \sqrt{{g_1}^2 + {g_2}^2 + \cdots + {g_t}^2}$$

= magnitude of the vector where every element of that vector is the gradient of the past t iterations for a particular weight (this is computed seperately for each weight/paramater)

Adagrad (adaptive gradient algorithm)

Main idea: A modified stochastic gradient descent algorithm with per-parameter learning rate

- Change the learning rate α based on past gradients of a particular weight



Mathematical concept:

For each iteration:

$$g_t = \nabla_\theta J(\theta_{t-1})$$

$$sum_t = sum_{i-1} + {g_t}^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\epsilon + sum_t}} \cdot g_t$$

Gradient at current iteration

Sum of all gradients

Magnitude of all past gradients

Small value to prevent division by zero

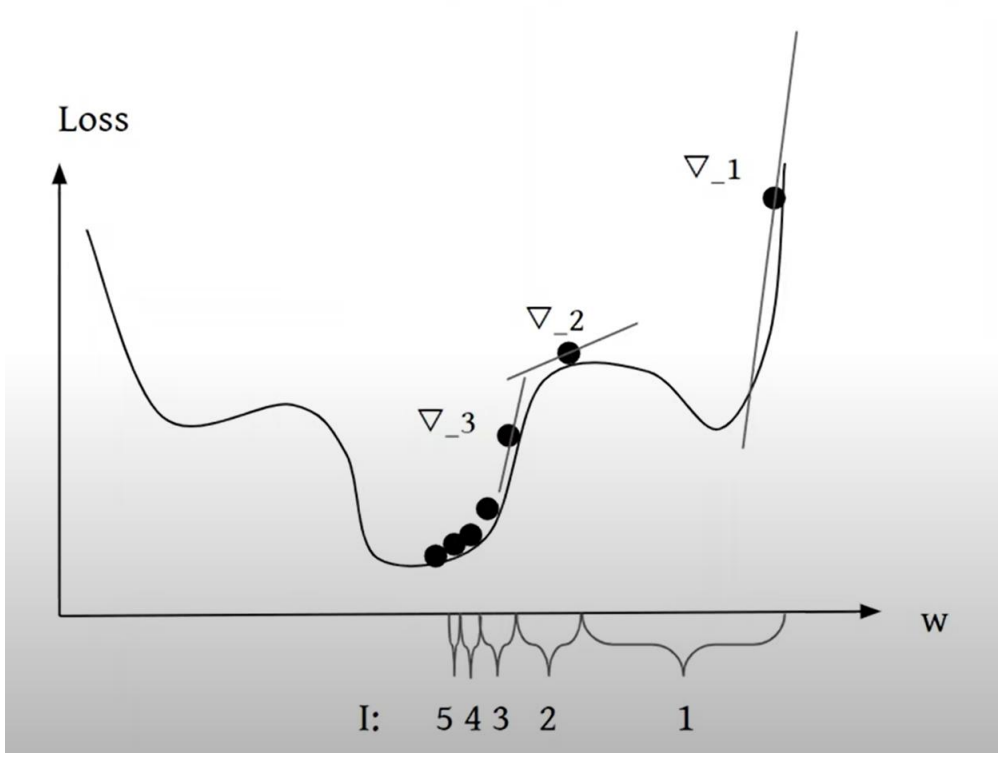
The larger the past gradients of a weight is, the smaller will be the learning rate is going to be.

Problem?

The learning rate decreases over time.

RMSProp

Main idea: Change the learning rate α based on past gradients of a particular weight with a decay rate β.



Mathematical concept:

For each iteration:

$$g_t = \nabla_\theta J(\theta_{t-1})$$

$$sum_t = \beta \cdot sum_{i-1} + (1-\beta) \cdot {g_t}^2$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\epsilon + sum_t}} \cdot g_t$$

Gradient at current iteration

Weighted sum of all gradients

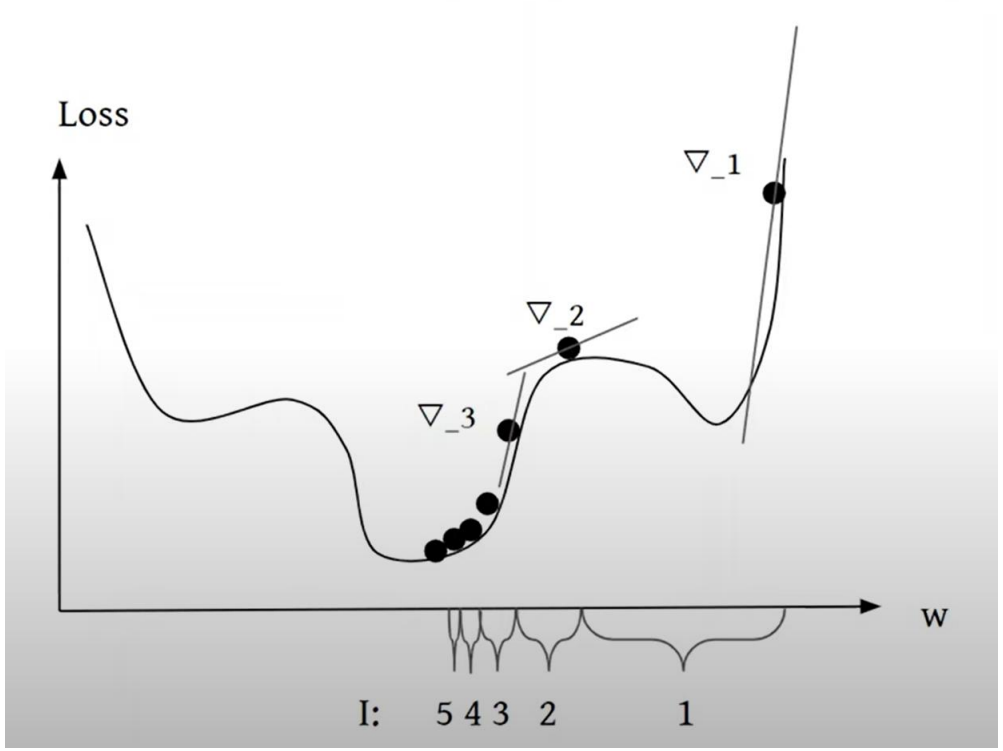
Magnitude of all past gradients

Small value to prevent division by zero

Note that the processes for RMSProp is identical to AdaGrad with the addition of decay.

Adam optimizer (Adaptive Moment Estimation)

Main idea: We want both momentum to give us a smooth weight update and adaptive learning rates



Momentum:

$$v_t = \gamma v_{t-1} + (1-\gamma) g_t$$
$$\theta_t = \theta_{t-1} - \alpha v_t$$

RMSProp:

$$sum_t = \beta \cdot sum_{i-1} + (1-\beta) \cdot {g_t}^2$$
$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\epsilon + sum_t}} . g_t$$

Adam:

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\epsilon + sum_t}} . v_t$$

Which optimizer should we use?

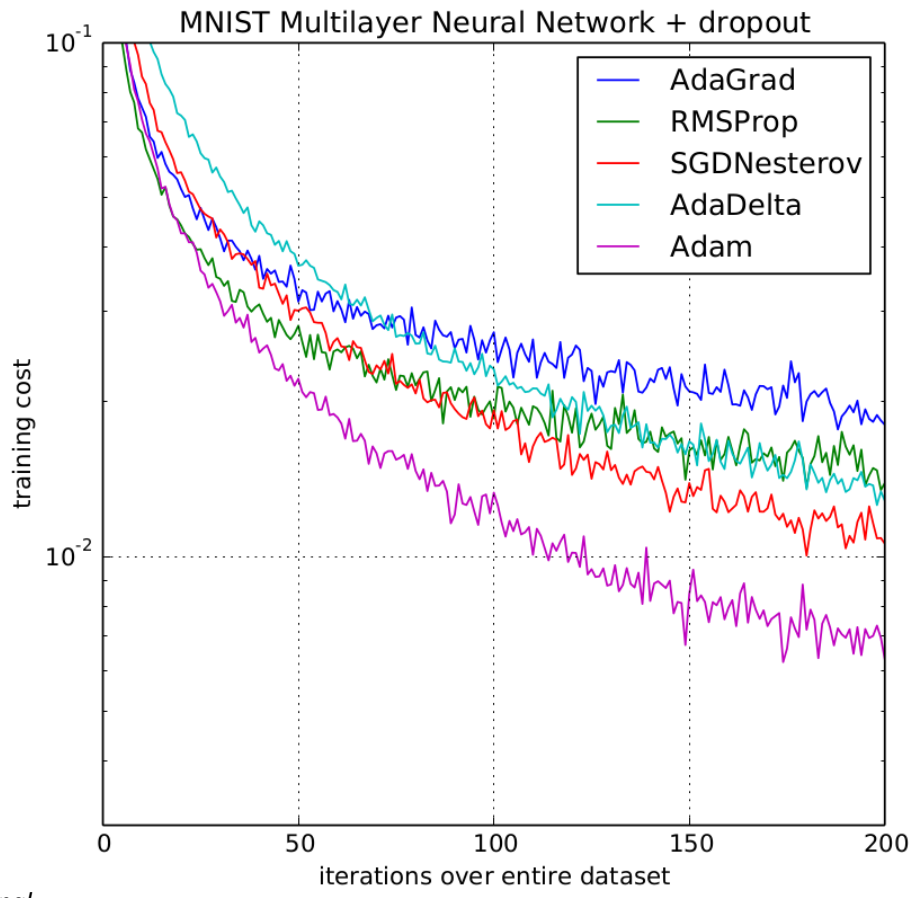
Adam works in areas with tiny gradients (such as saddle points or ravines). In these areas SGD struggles to quickly navigate through them.

Adam converge faster than SGD.

Still, state-of-the-art results in classification tasks, including CIFAR-10 are achieved using good old SGD.

Recent studies have shown that Adam and other adaptive gradient methods do not generalize as well as SGD with momentum [1,2] and that SGD is more stable for strongly convex loss functions [3].

One argument is that faster convergence leads to fewer epochs and, therefore, the model has not seen all the data several times.

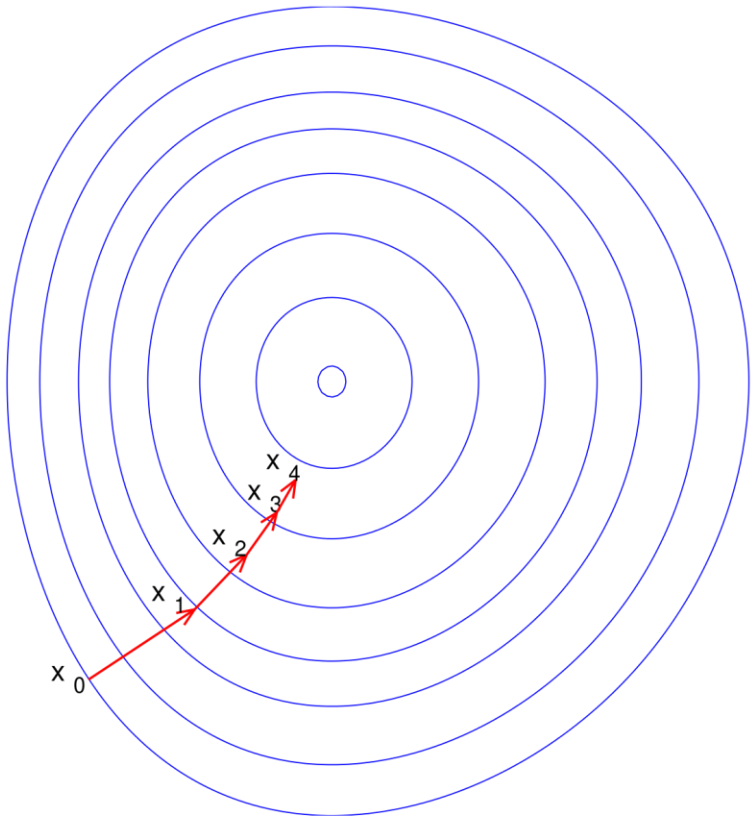


[1] Wilson, Ashia C., Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. "The Marginal Value of Adaptive Gradient Methods in Machine Learning." *ArXiv:1705.08292 [Cs, Stat]*, May 21, 2018. http://arxiv.org/abs/1705.08292.
[2] Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. arXiv preprint, arXiv:1904.09237.
[3] Hardt, M., Recht, B., & Singer, Y. (2016, June). Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning* (pp. 1225–1234). PMLR.

Image from Kingma, Diederik P., and Jimmy Ba. "Adam: A Method for Stochastic Optimization." *ArXiv:1412.6980 [Cs]*, 2014.

Learning rate decay

- In training deep networks, it is usually helpful to anneal the learning rate over time if you are not using an adaptive optimizer.
- A high learning rate can help us escape local minima and approach good weights, but can make us bounce around chaotically, unable to settle down into deeper parts of the loss.
- A strategy is, therefore, to start with a high learning rate, and reduce the learning rate over time



Learning rate decay

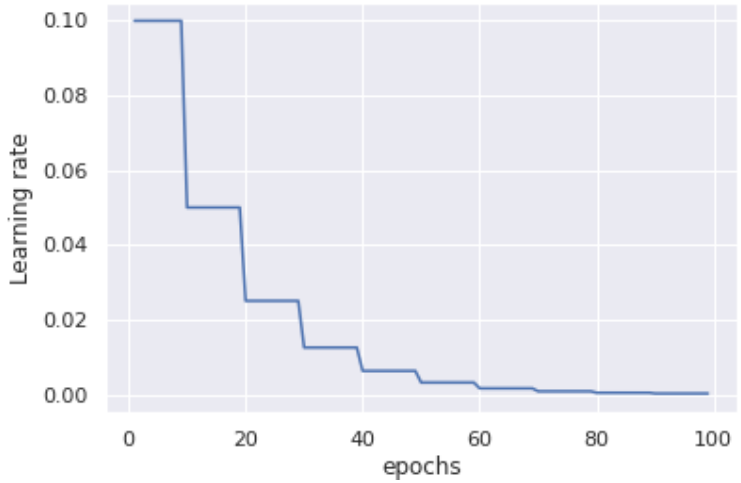
Step decay:

- Reduce the learning rate by some factor every few epochs. A typical factor is 0.1 to 0.5
  - The number of epoch depends on the data and model size, but it could be, e.g., every 1/5 of the total number of epochs
- A practical way to implement step-decay is to reduce the learning every time the validation error stops improving (sometimes referred to as "update on plateau")

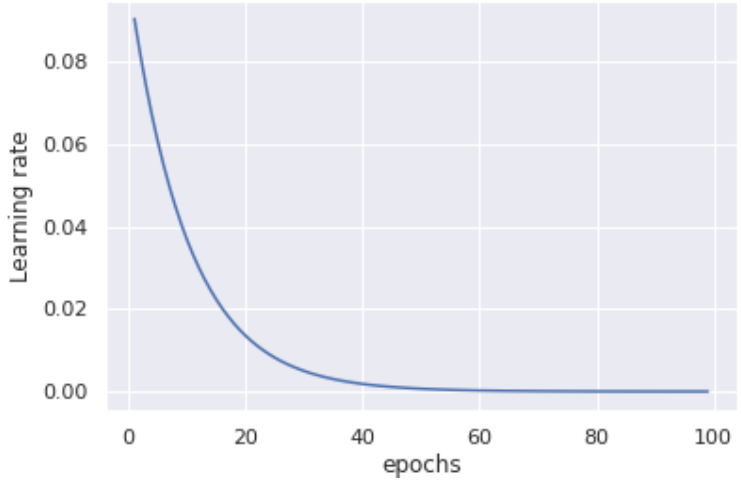
Exponential decay:

has the mathematical form $\alpha=\alpha_0 e^{-kt}$, where α_0, k are hyperparameters and t is the iteration number (but you can also use units of epochs).

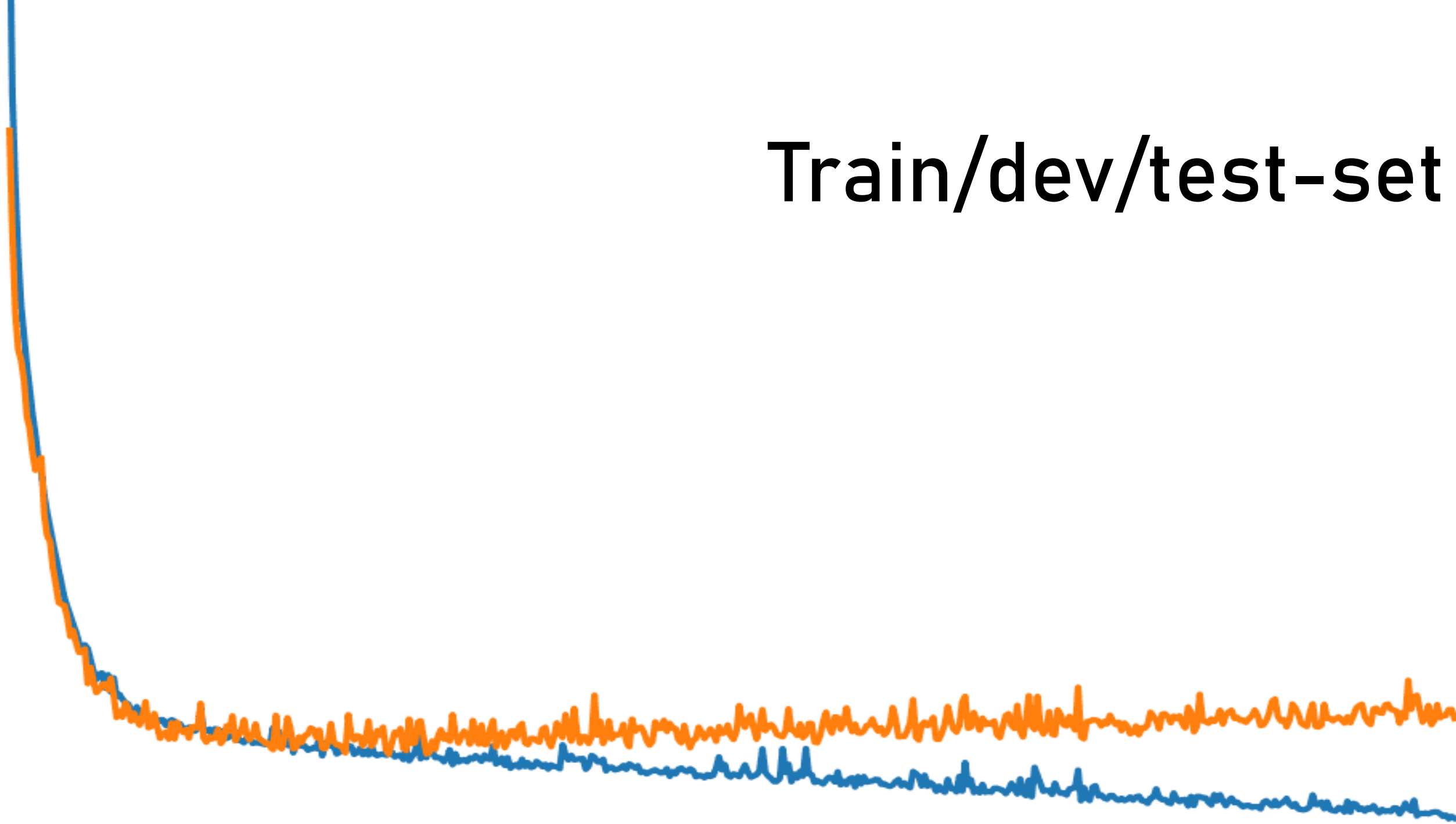
The LR is reduced by a factor of 0.5 every 10 epochs



The LR is reduced by $e^{-0.1t}$



Train/dev/test-set



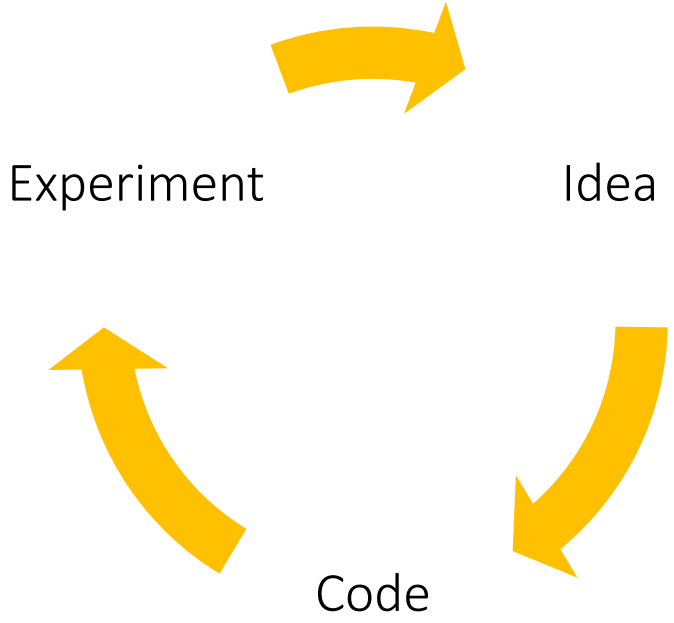
Training, validation and Test-set?

- The **training data set** is a set of examples used to fit the parameters (e.g. weights of connections between neurons in artificial neural networks) of the model.
- The **validation data set** provides an unbiased evaluation of a model fit on the training data set while tuning the model's hyperparameters (e.g. the number of hidden units—layers and layer widths—in a neural network)
- the **test data set** is a data set used to provide an unbiased evaluation of a *final* model fit on the training data set.

- In some litterature, the term " development set"
  is sometimes used instead of ”validation set", while in other litterature, the term "validation set" is used instead of "test set".

How to choose training, development and test-set?

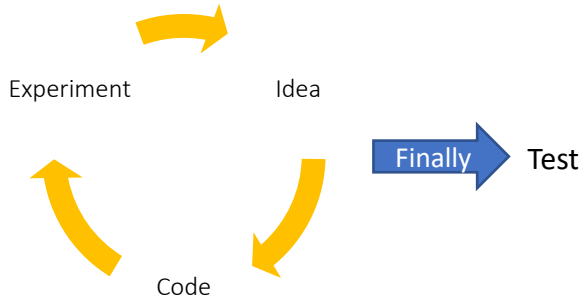
- Designing a deep neural network is an iterative process, where hyperparameters are changed.
  - # layers
  - # hidden units
  - Activation functions
  - Learning rate
  - Optimizer
  - …



It is almost impossible to guess the best hyperparameters and hard to choose good hyper parameters in the first attempt

Train, development, and test set

- We Iterate between training and validation while tuning the model's hyperparameters.
- If we use the test set (rather than the validation set) when selecting hyperparameters, it's possible we'll end up overfitting our hyper-parameters to the test data.
  - Therefore, we should not use the test set before we have settled on a model.



Train, development, and test set split ratios

- With a small training data set, you could choose to split the data in
  80% train / 10% validation / 10% test
- With a very large data set (e.g. 1'000'000 samples), you can have a smaller percentage for the development and test set. E.g.
  98% train / 1% validation / 1% test (that's still 10'000 samples for validation)

- Remember, the point of the development set is just to choose between different hyperparameters.

The larger the dataset, the smaller the validation set and vice versa.

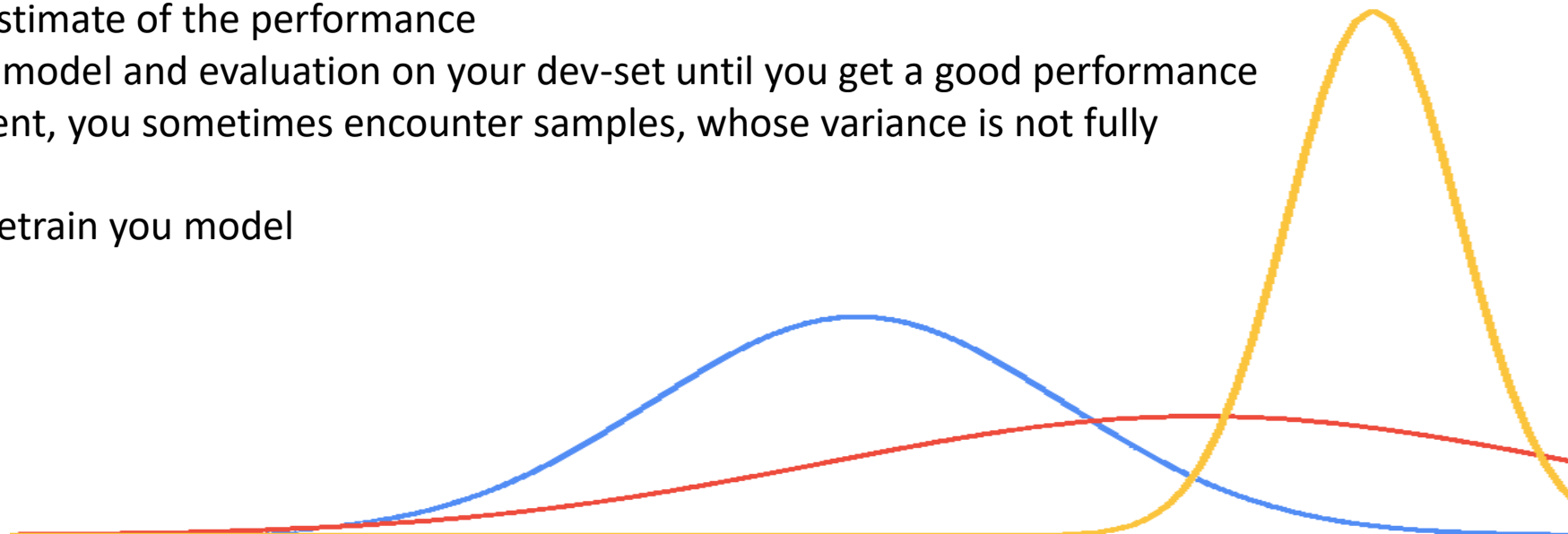
You should ensure, that the data variation is covered in each data set. I.e. the Train and Test distribution should be matched. At least test and validation set should be matched

Train and test set distribution should be matched

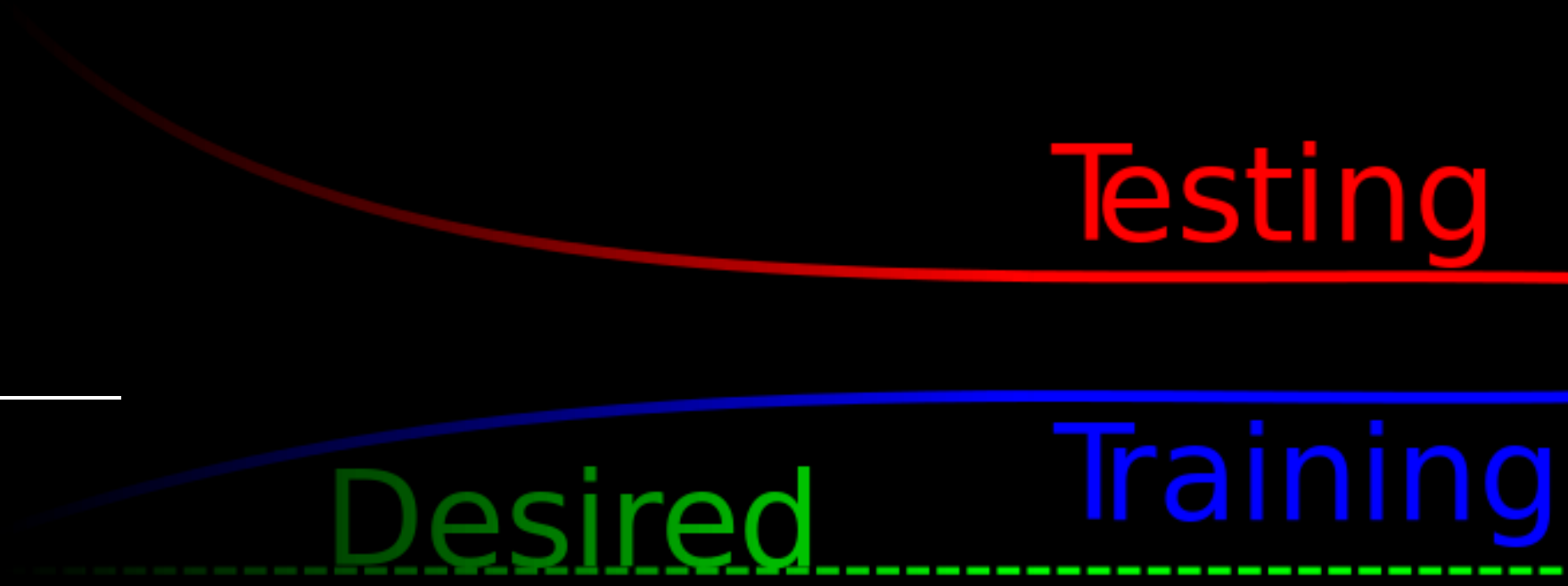
- When developing your model, you should make sure that at least the val. and test-set come from the same distribution, without information lekage.

Do we always need a test set?

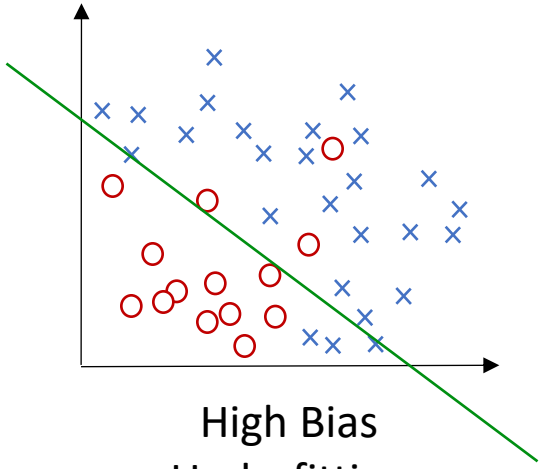
- Only If you need unbiased estimate of the performance
- You can iteratively train you model and evaluation on your dev-set until you get a good performance
- In practice, during deployment, you sometimes encounter samples, whose variance is not fully covered in your training-set
    - You may then have to retrain you model



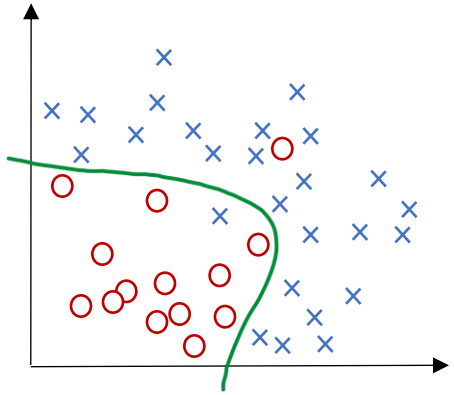
Bias and variance



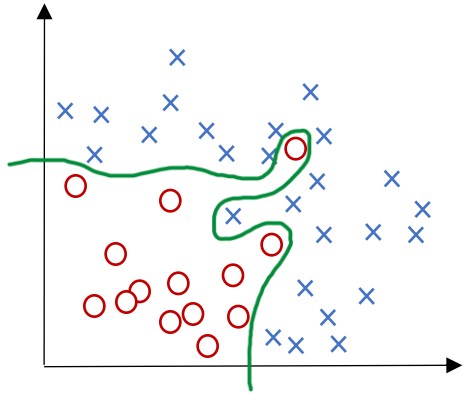
Bias and variance



High Bias
Underfitting



Just right



High variance
Overfitting

We can use **bias** and **variance** to guide us while training

In 2D it is easy to see from the decision boundary, if we are overfitting or underfitting.
We need other tricks in higher dimentions

Understanding bias and variance:

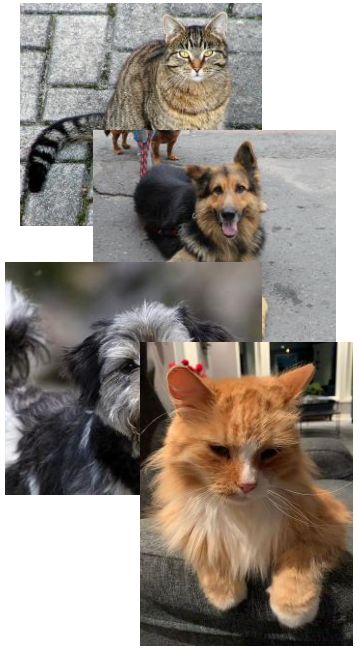
We evaluate the performance using

- Train set error
- Development set error

Bias: How big is the error when evaluating the model on your training Set?

Variance: How big is the error when evaluating the model on your dev. set compared to the error on the train-set?
(How well does your model generalize from training to evaluation?)

Example: Cat and dog classification with perfect labelled data (0% error in labels)

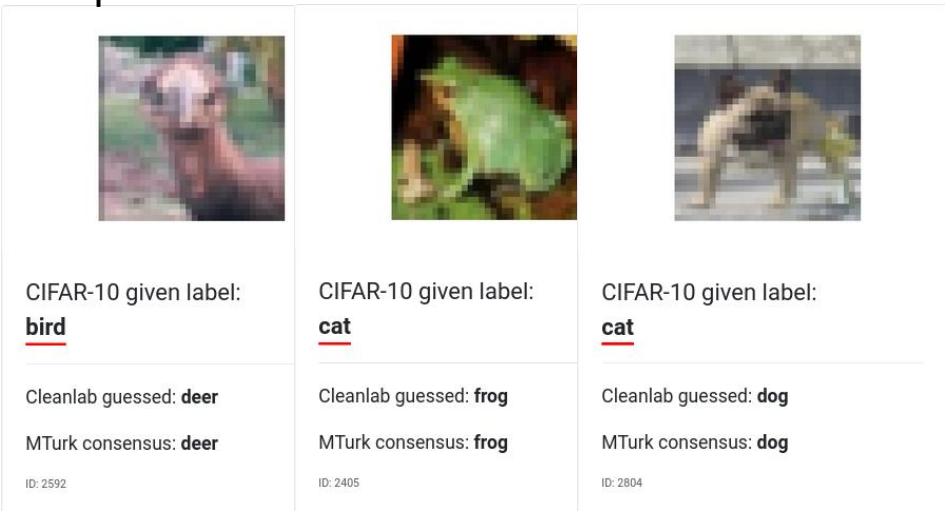


Train set error: 1% Dev set error: 10%	Small difference between the label error (0%) and the train set error --> **low bias** big difference between the error rate of train and dev. set --> **high variance**
Train set error: 17% Dev set error: 18%	big difference between the label error (0%) and the train set error --> **high bias** small difference between the of train and dev. set error--> **low variance**
Train set error: 17% Dev set error: 30%	big difference between the label error (0%) and the train set error --> **high bias** Big difference between the train and dev. set error --> **high variance**
Train set error: 1% Dev set error: 2%	Small difference between the label error (0%) and the train set error --> **low bias** small difference between the train and dev. set error --> **low variance**

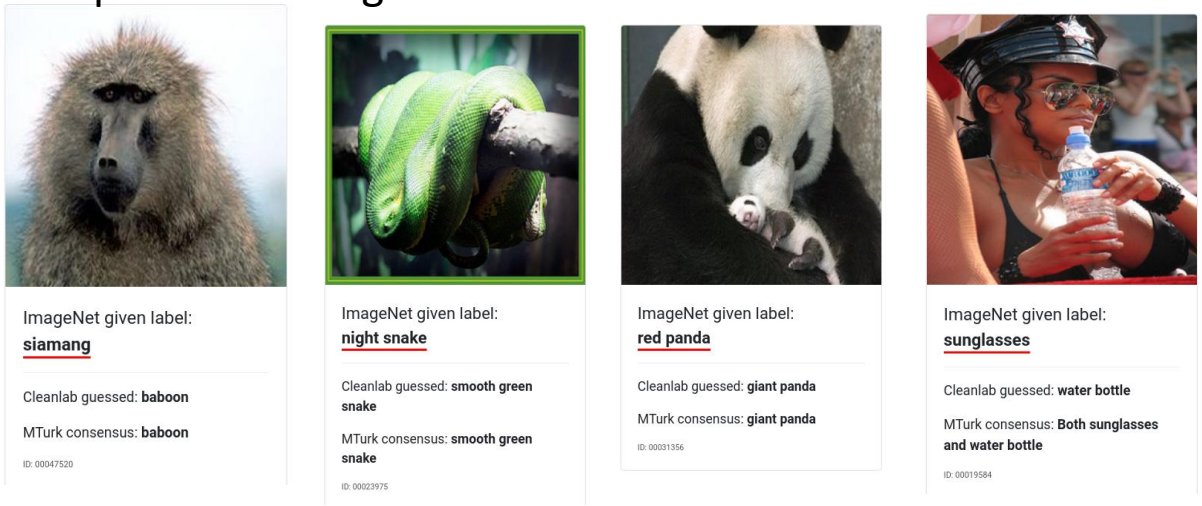
Bayes error: The error in the labels

- In the cat/dog example we assumed that the labels were perfect. However, that is not always the case. The bias should therefore be seen in the light of the *Bayes error*.
- It is the **lowest possible test error rate** in classification which is produced by any classifier.
- If the Bayes error is 17%, a train set error 17% would be a low bias.
- Most of the famous big data sets have errors. Take a look at https://labelerrors.com/

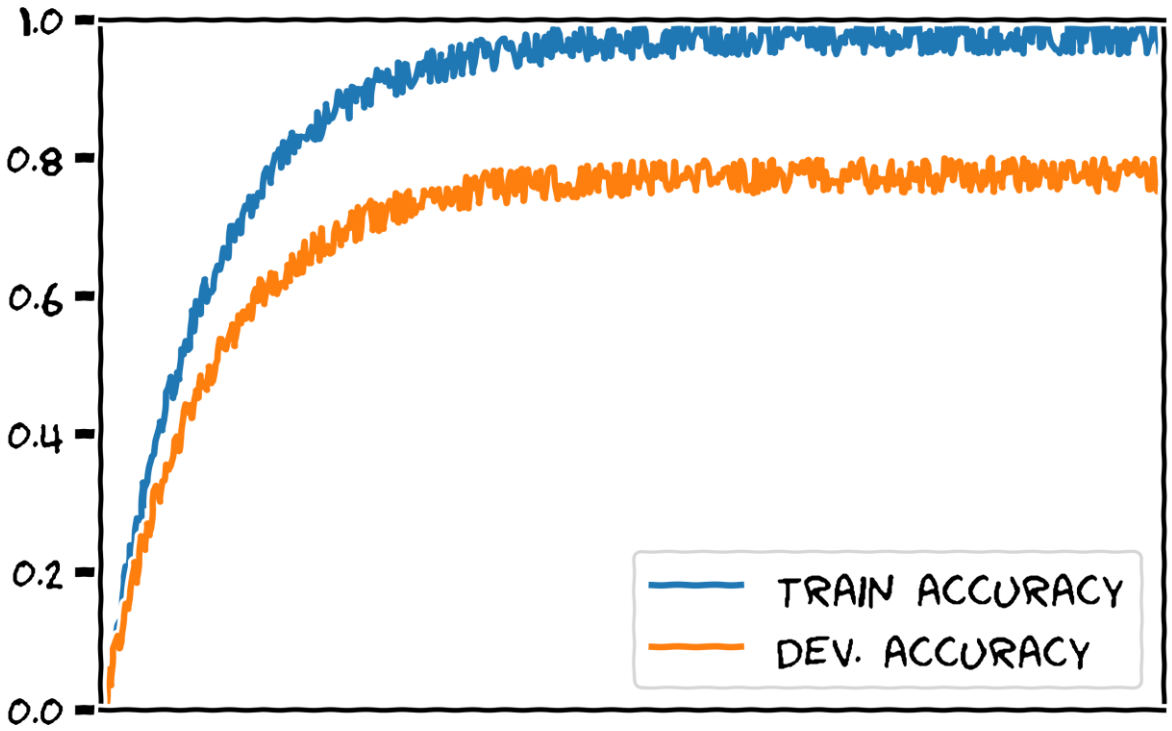
Samples from Cifar-10



Samples from ImageNet



QUIZ: Bias & Variance



How Would you characterize these errors?
(label error is 0%)

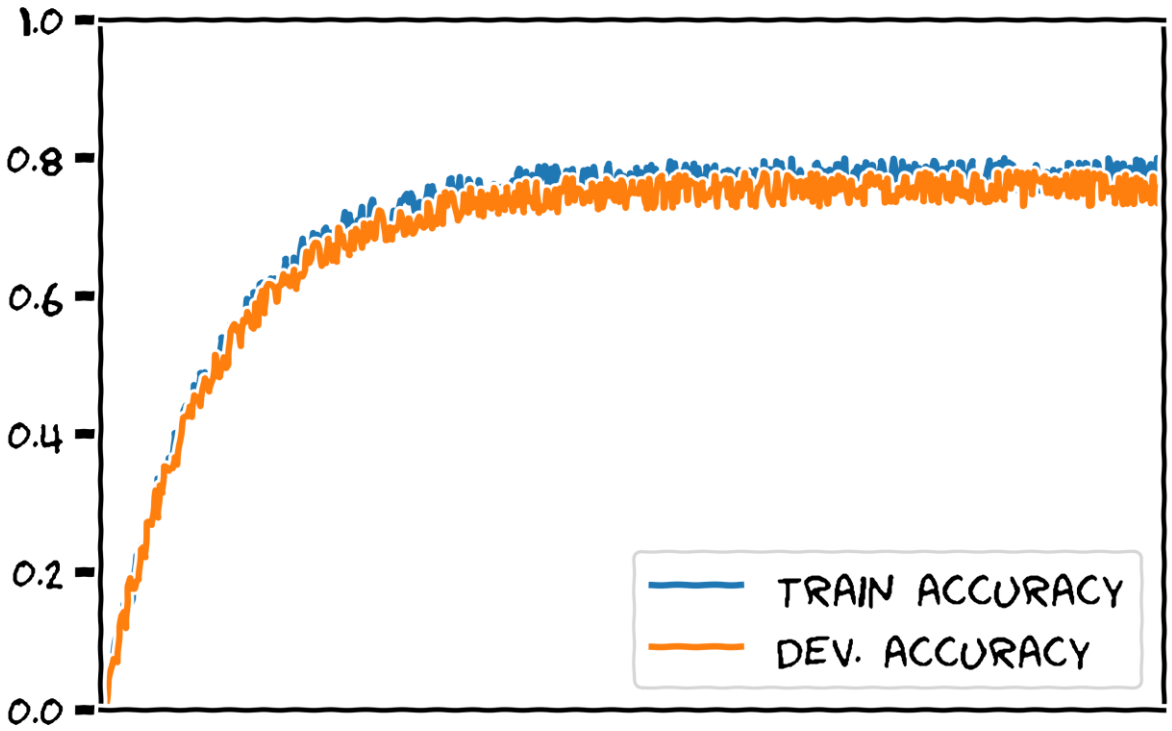
A
Low Bias
Low Variance

B
Low Bias
High Variance

C
High Bias
Low Variance

D
High Bias
High Variance

QUIZ: Bias & Variance



How Would you characterize these errors?
(label error is 0%)

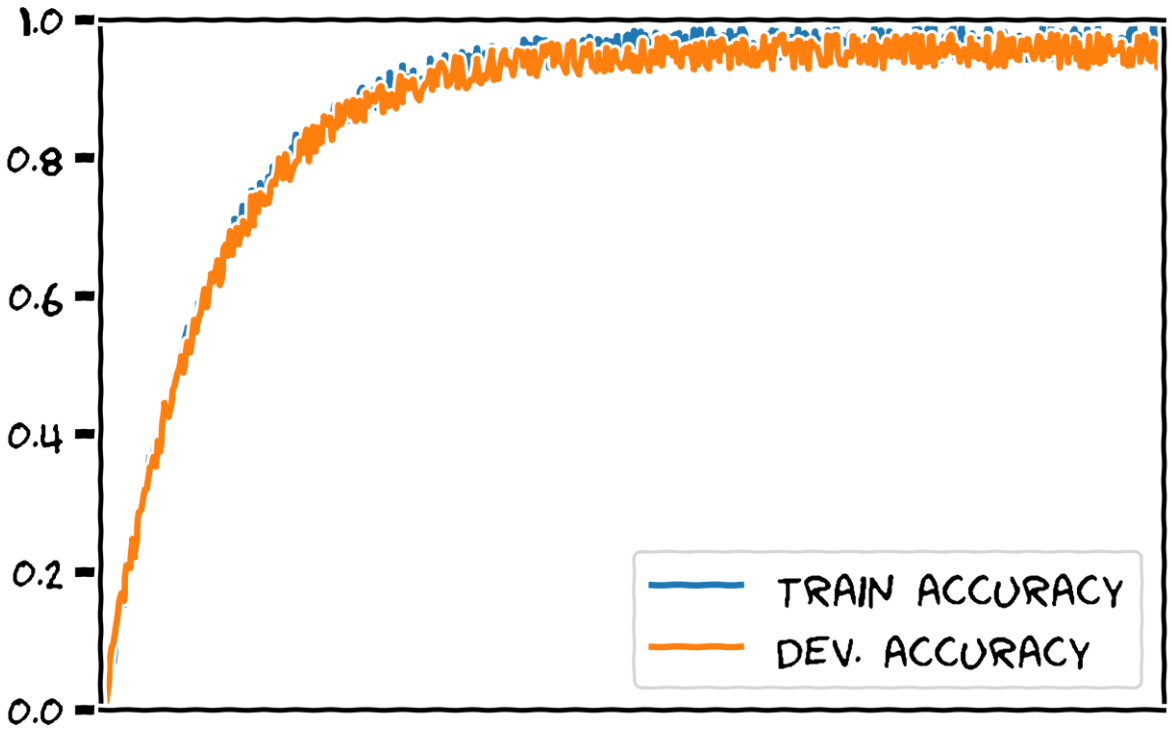
A
Low Bias
Low Variance

B
Low Bias
High Variance

C
High Bias
Low Variance

D
High Bias
High Variance

QUIZ: Bias & Variance



How Would you characterize these errors?
(label error is 0%)

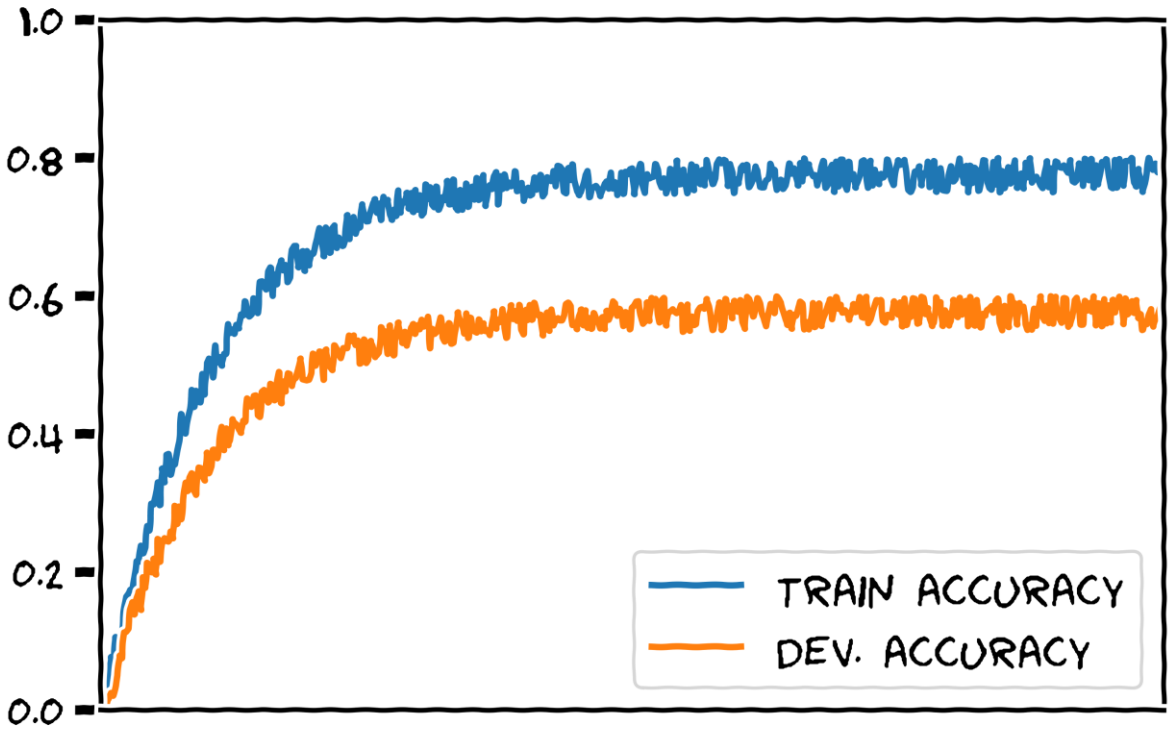
A
Low Bias
Low Variance

B
Low Bias
High Variance

C
High Bias
Low Variance

D
High Bias
High Variance

QUIZ: Bias & Variance



How Would you characterize these errors?
(label error is 0%)

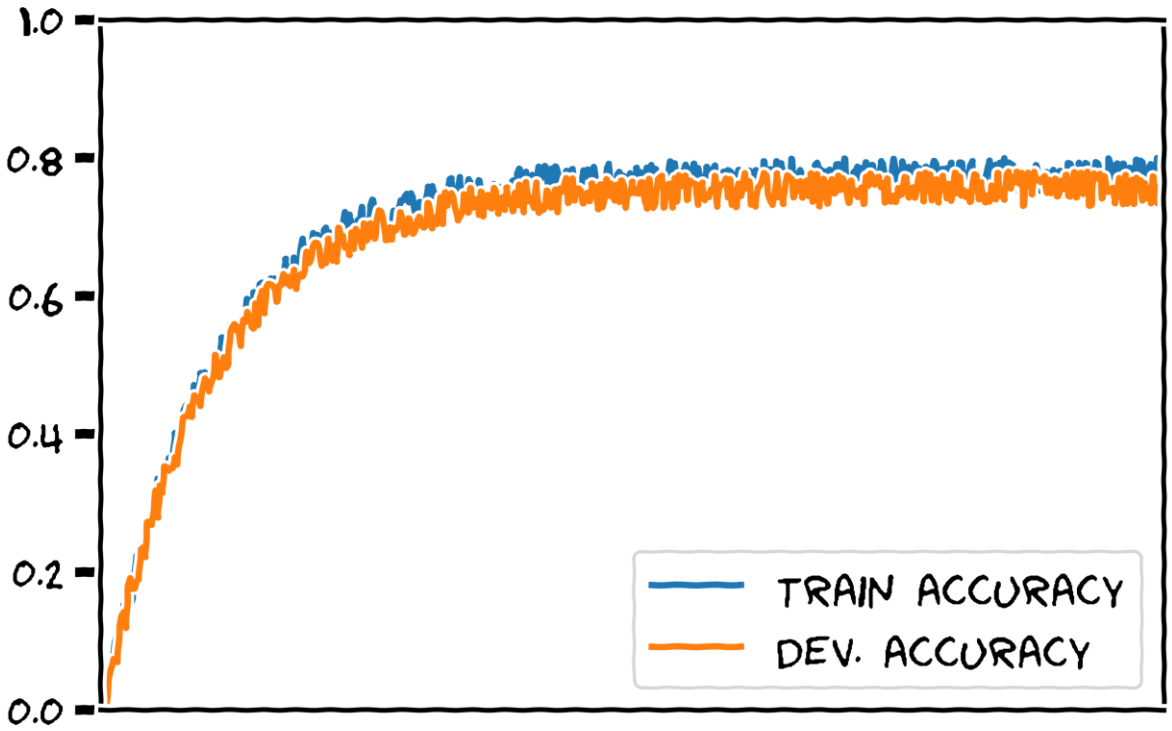
A
Low Bias
Low Variance

B
Low Bias
High Variance

C
High Bias
Low Variance

D
High Bias
High Variance

QUIZ: Bias & Variance



How Would you characterize these errors?
(label error is 20%)

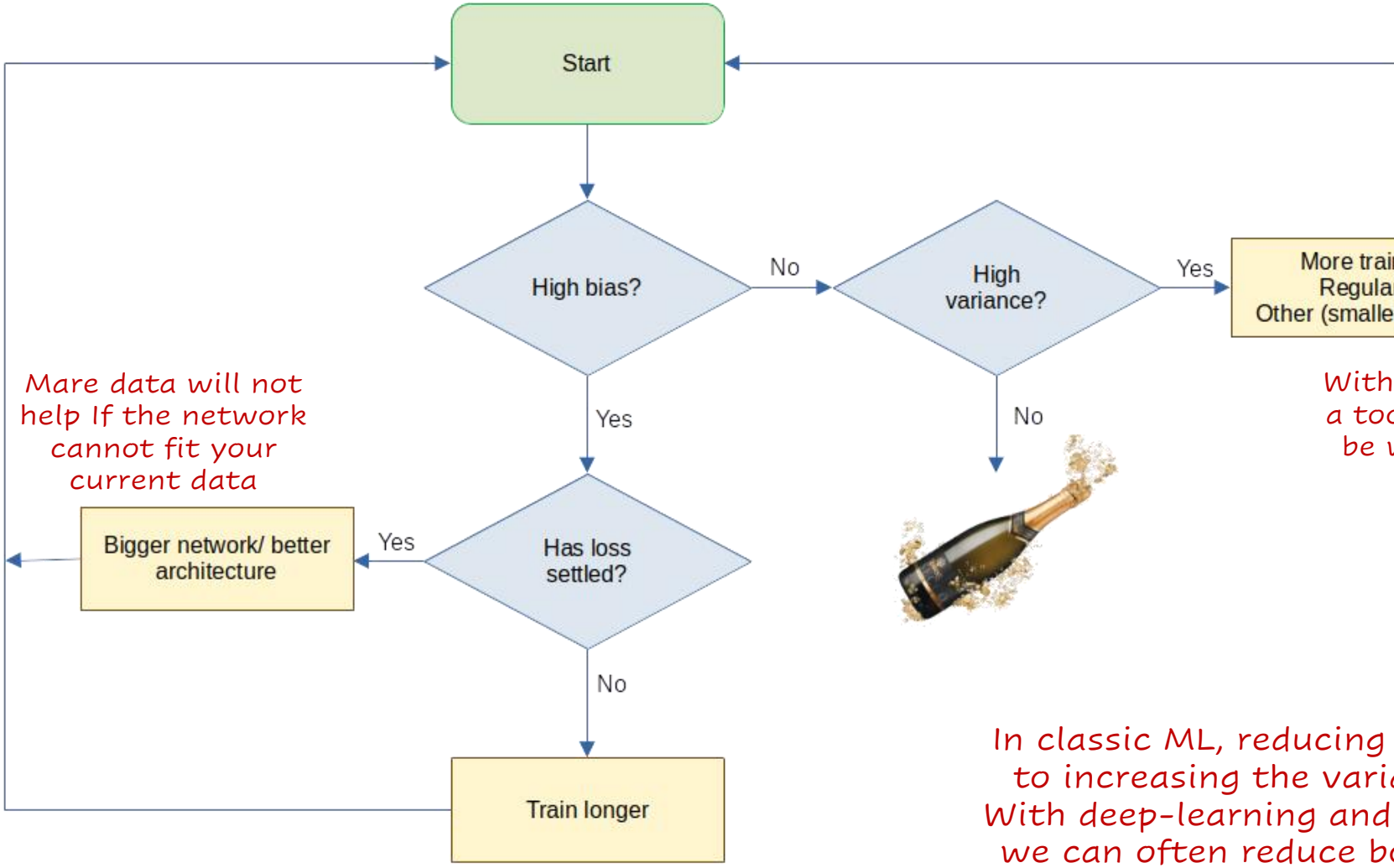
A
Low Bias
Low Variance

B
Low Bias
High Variance

C
High Bias
Low Variance

D
High Bias
High Variance

The deep-learning training flow-chart



You are probably overfitting your network

With proper regularization, a too big network will just be wasted computations

Mare data will not help If the network cannot fit your current data

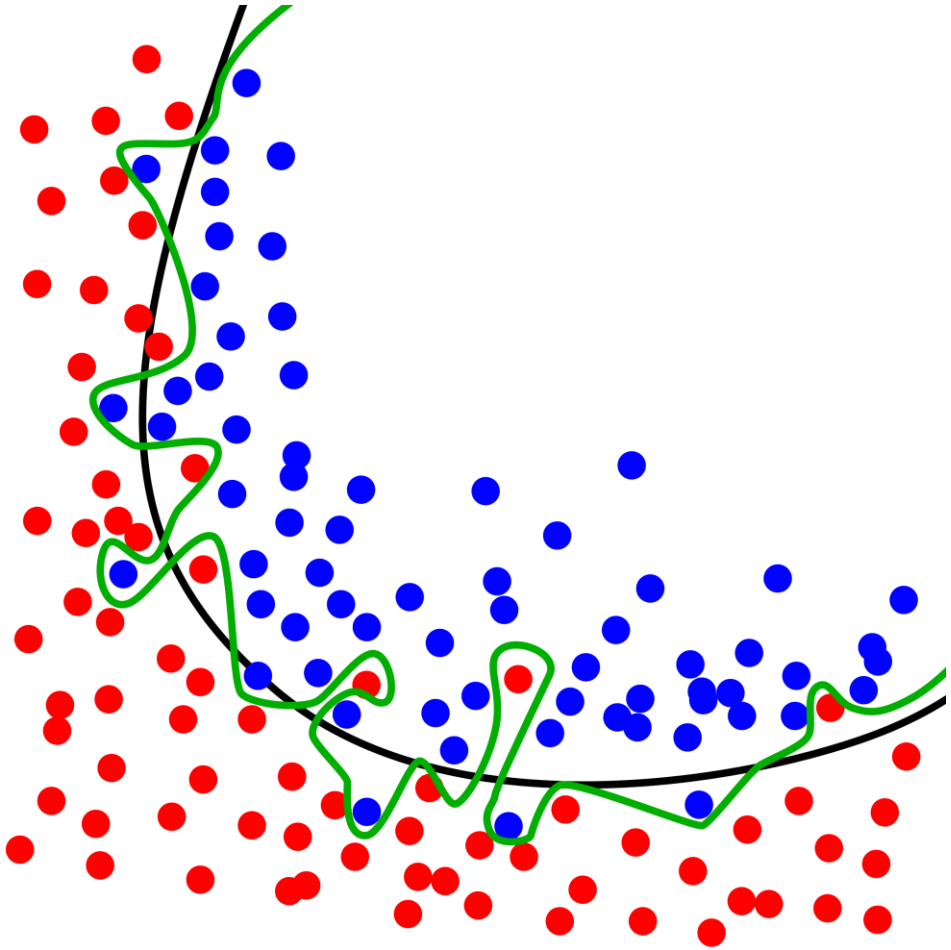
In classic ML, reducing bias would often lead to increasing the variance, and vice versa. With deep-learning and proper regularization, we can often reduce both bias and variance at the same time.

Regularization

Our goal is to discover patterns. Regularization helps us discover a general pattern and not simply memorized our data.

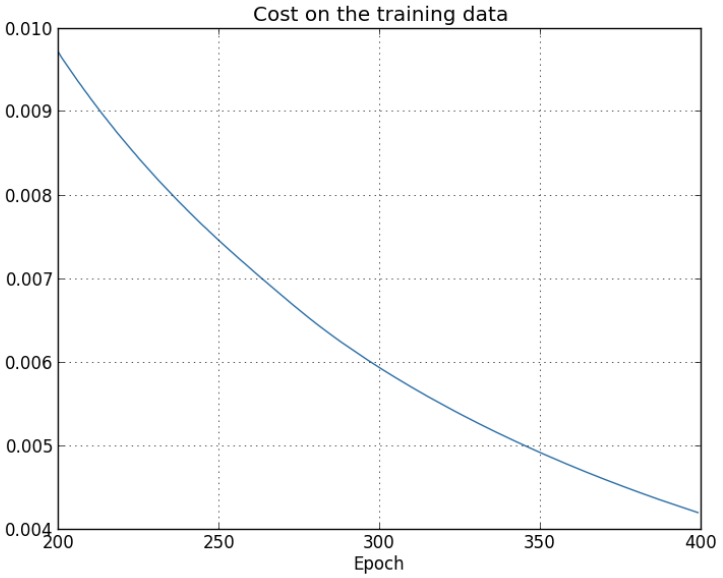
Regularization

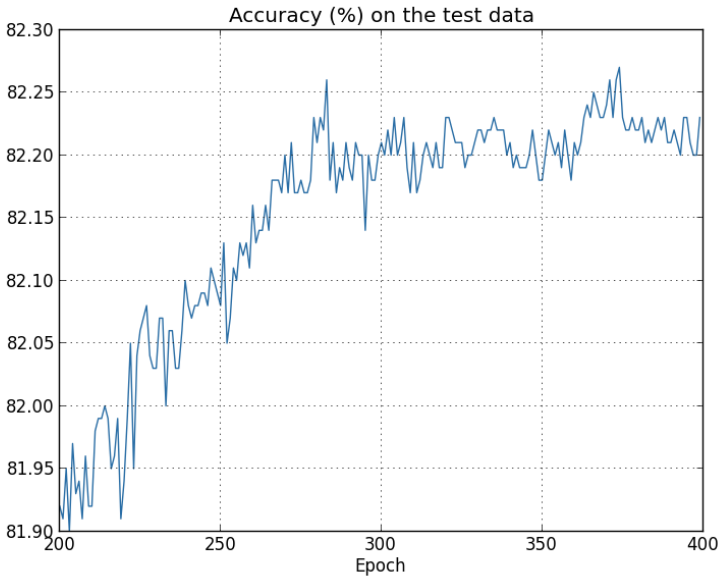
- If you have a big gap between the performance on the **development data** and the **training data** (variance problem / overfitting), we might need some regularization.
- Regularization helps reduce overfitting
  - Penalize complexity to improve generalization



Overfitting MNIST

- The figures show cost and accuracy on 1000 training images from MNIST, when training a 30 hidden neuron network (23,860 parameters)
- Notice how the cost keeps decreasing smoothly, while the accuracy on the test data settles after 275 epochs
- After epoch 275, the network no longer generalizes to the test data. We say that the network is overfitting.
- A large network with many parameters will overfit more easily, as the network has the capacity to recognize the training data rather than "understand" the data.
    - We could reduce the size of the network or add regularization





Images from "Neural Networks and Deep Learning" by Michael Nielsen

L2 regularization (Euclidian norm regularization)

- One of the most common types of regularization
- Penalizes large weights

Example: Logistic regression

We want to minimize the cost function, J

$$min_{w,b} J(w,b)$$

$$J(w,b) = \underbrace{\frac{1}{m}\sum_{i=1}^{M} L(\hat{y}^i, y^i)}_{\text{Data loss: predictions should match training data}} + \underbrace{\frac{\lambda}{2m}\|w\|_2^2}_{\text{L2 Regularization: Prevent the model from doing too well on training data}}$$

$$w \in \mathbb{R}^{n_x}$$
$$b \in \mathbb{R}$$

i is the individual samples,
n_x is the individual weights
m is the size of the training set
L is our loss function, e.g., Cross entropy or quadratic cost
λ (>0) is the regularization strength

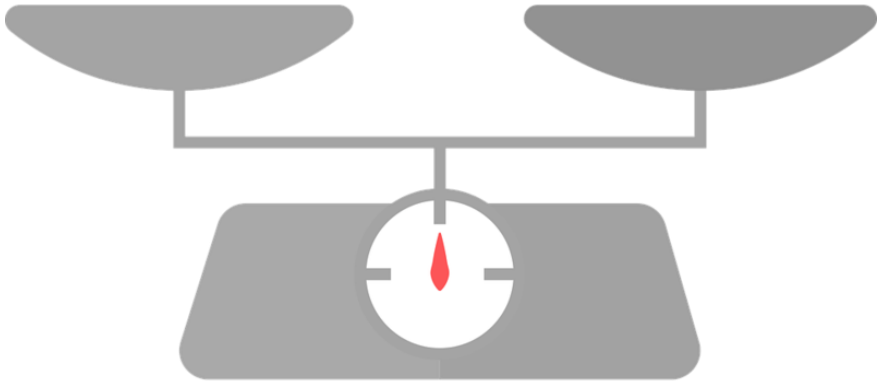
$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

We could also add regularization based on the bias ($\frac{1}{2m}b^2$), but the impact is small since it is only a single value, whereas the impact from *w* is much larger, since it is high-dimensional

L2 regularization Intuitions

- Many parameters can lead to overfitting. We would, therefore, prefer the network to be sparse.
- Large weights will only be allowed, if they considerably reduce the original (blue) cost function
  - We are making a compromise between minimizing the original cost function and finding small weights

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{M} L(\hat{y}^i, y^i) \quad + \quad \frac{\lambda}{2m}\|w\|_2^2$$



L2 regularization in neural networks (Weight decay)

In neural networks you have multiple weights. Therefore, the L2 regularization becomes the sum of all your weights

layers

$$J(w^{[1]}, b^{[1]}, ..., w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{M} L(\hat{y}^i, y^i) + \frac{\lambda}{2m}\sum_{l=1}^{L} \left\|w^{[l]}\right\|^2$$

$$\left\|w^{[l]}\right\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$$

The rows "i" of the weight matrix is the number of neurons in the current layer $n^{[l]}$
The columns "j" of the weight matrix should equal the number of neurons in the previous layer $n^{[l-1]}$.

Gradient descent weight update with L2 regularization

$$\frac{\partial J}{\partial w^{[l]}} = \{\text{from backprob}\} + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} := w^{[l]} - \alpha \cdot \frac{\partial J}{\partial w^{[l]}}$$

Learning rate

$$\frac{\partial J_0}{\partial w^{[l]}}$$

Gradient from backprob.

$$w^{[l]} := w^{[l]} - \alpha \left(\{\text{from backprob}\} + \frac{\lambda}{2m} w^{[l]} \right)$$

$$w^{[l]} := w^{[l]} - \alpha \left(\frac{\lambda}{2m} w^{[l]} \right) - \alpha\{\text{from backprob}\}$$

L2 norm reduces the weights and therefore it is also called "Weight decay"

L1 regularization

- Like the L2 regularization, L1 penalizes large weights. However, while the L2 norm shrinks weights proportionally to their value, L1 norm shrinks weights with a fixed value.
  - L2 penalizes large values more than L1
  - L1 penalizes small values more than L2
- L1 encourages model coefficients to shrink to zero
  - L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connection
    - We say that the weight matrices are sparse
- L1 regularizations is not used very often, compared to L2 regularization

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{M} L(\hat{y}^i, y^i) \quad + \quad \frac{\lambda}{m}\sum_{l=1}^{L}\left|w^{[l]}\right|$$

L2 regularization in Pytorch

- The SDG optimizer has a parameter called "weight_decay" that corresponds to L2-regularization with *2 * lambda.*
- Other optimizers (e.g. Adam) have similar functionality that takes into account the update rate of a given dimention.

You can implement L2 manually this way:

```
outputs=model(inputs)
loss=loss_fn(outputs,labels)

l2_lambda = 0.001
l2_norm = sum(p.pow(2.0).sum()
                  for p in model.parameters())

loss = loss + l2_lambda * l2_norm
```

Optimizers and Regularization in PyTorch

You can find a list of optimizers already implemented in PyTorch at this link:

https://pytorch.org/docs/stable/optim.html

E.g.

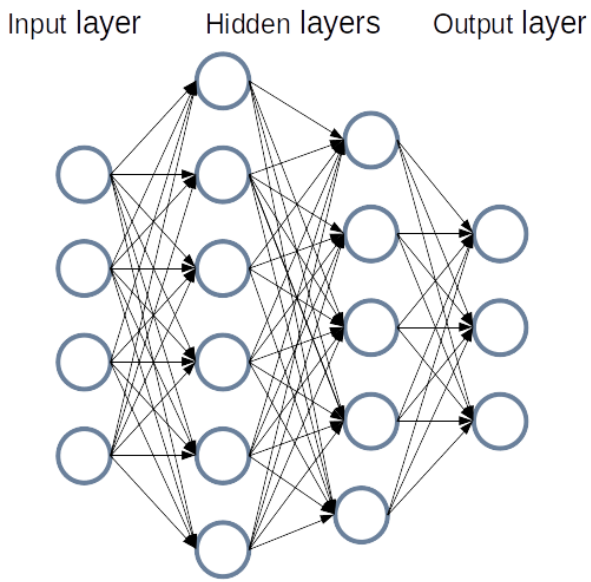
```
torch.optim.SGD(params, lr=<required parameter>, momentum=0,
     dampening=0, weight_decay=0, nesterov=False)

torch.optim.Adam(params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0,
     amsgrad=False, ….)
```

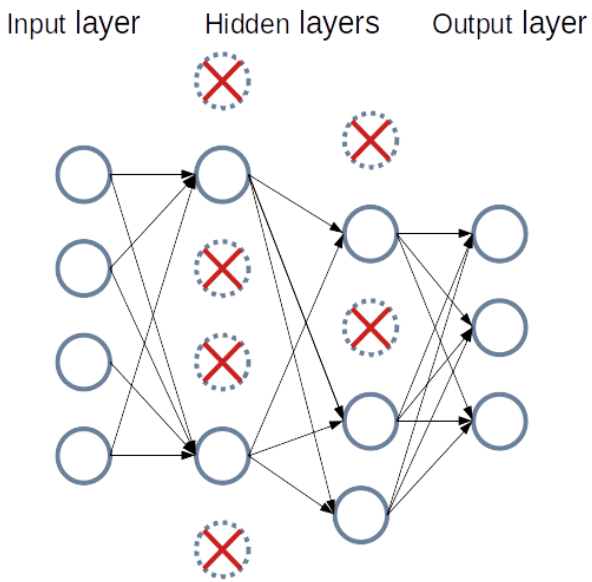
Dropout regularization

- Dropout has been useful in training large, deep networks, where the problem of overfitting is often acute.
- In dropout, we modify the network instead of the cost function
- In each forward pass random neurons are disabled until the next forward pass
- Often used in fully-connected layers, where it helps balancing the number of parameters with the conv-layers
- Dropout reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Can also be seen as a way of creating an average of many neural networks

Without Dropout



Dropout

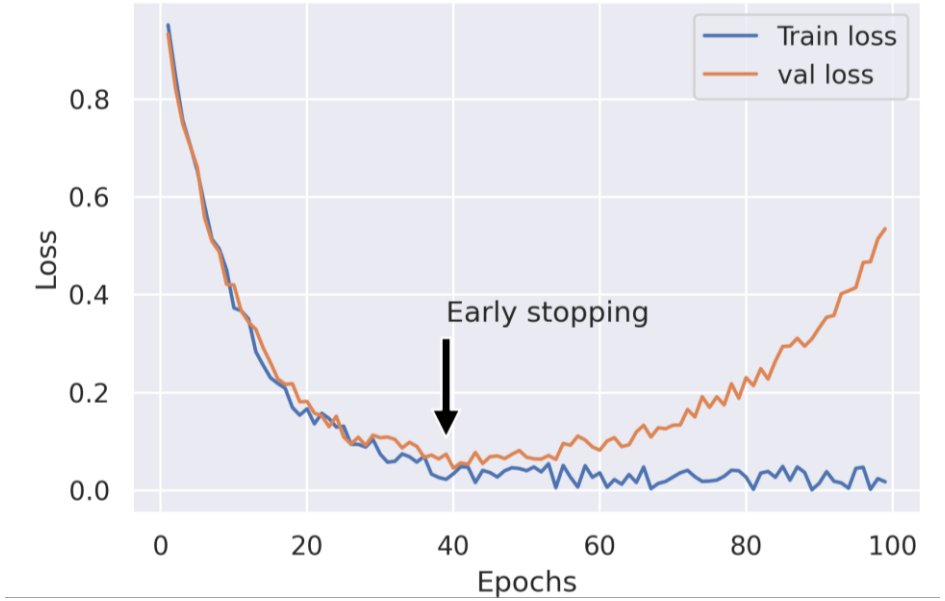


torch.nn.Dropout(p=0.5, inplace=False)

Remember to disable dropout after training

Early stopping

- Too many epochs can lead to overfitting whereas too few may result in an underfit model.
- Early stopping is a technique, where you specify too may epochs and keep track of the validation loss.
- You stop training when the validation starts increasing, I.e., you start overfitting
- A practical way of implementing Early stopping is by saving the weights if the current validation loss is less than the previous best one



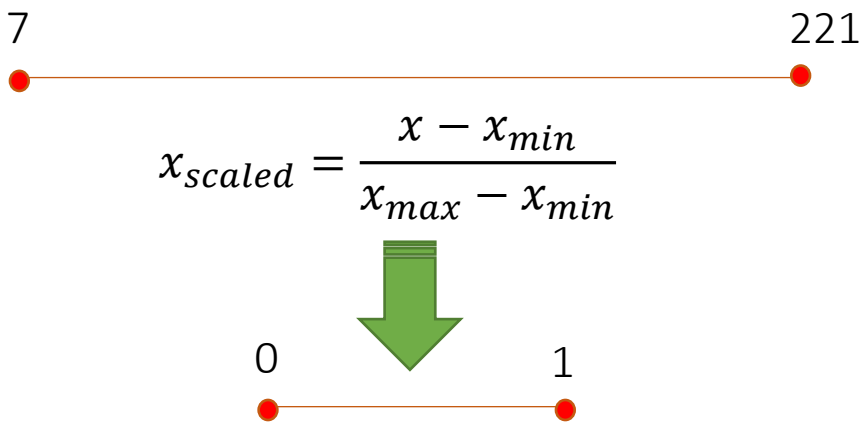

```
# Pseudo code
For e in range(n_epochs):
   Train()
   Val_loss = eval()

   if Val_loss < best_val_loss:
       best_params = params

Save_params(best_params)
```

Batch normalization

- Objective: we want to put our data on some known scale
- Large variations in features' scale can cause large difference in size of gradients, leading to instability.



$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Normalization: Scale to interval [0;1]

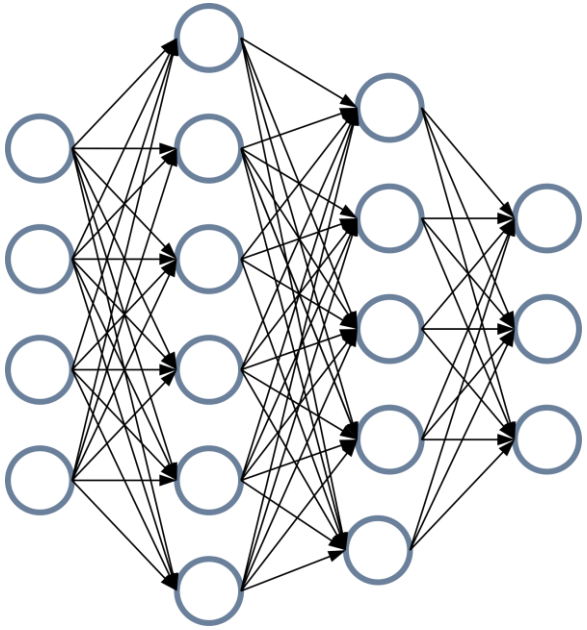
$$x_{standard} = \frac{x - \mu}{\sigma}$$

$$\mu = \frac{\sum x_i}{N} \qquad \sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

Standardization: Zero mean and std. diviation of 1

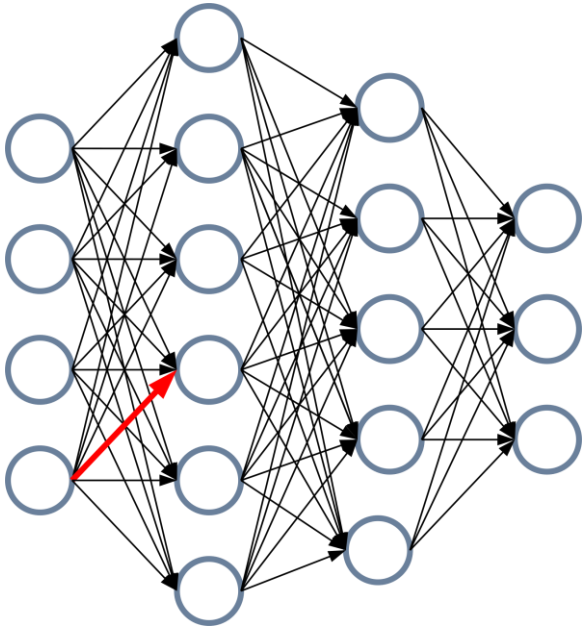
In batch normalization we are using standardization!

Batch normalization



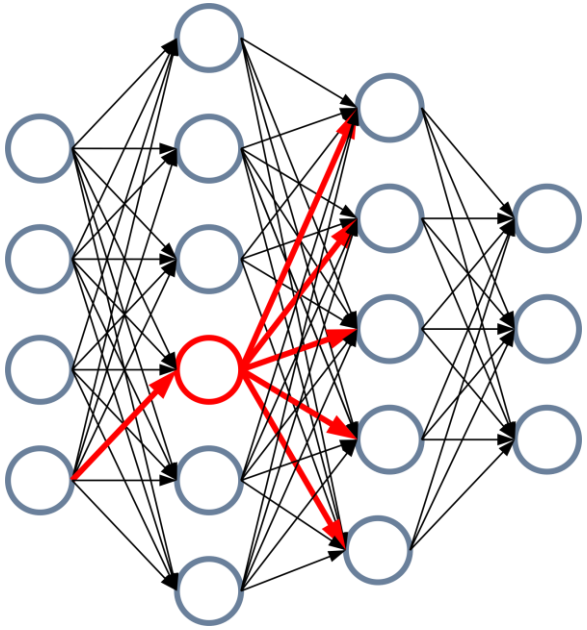
Batch normalization

If one weight becomes drastically larger than other weights



Batch normalization

If one weight becomes drastically larger than other weights, It will cause the output of the neuron to also be large, which can cause instability



Batch normalization

Batch normalization can be applied to single layers to set the range of the activations.

Use *B* to denote a mini-batch of size *m* of the entire training set. The mean and variance of the feature-maps z_i of mini-batch *B* could thus be denoted as

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} z_i \qquad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i - \mu_B)^2$$

The normalized output is then given by

$$\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Small number to ensure numerical stability

Now the output has zero mean and unit variance. It might, however, make sense for the network to represent other distributions. Therefore, a transformation step then follows as

$$\tilde{z}_i = \gamma \hat{z}_i + \beta$$

Learnable parameters that would be updated during training, like other weights in the network. (they are unrelated to gamma and beta from previous slides)

This means that the mean and std. deviation of y_i can be whatever we want them to be. The mean and variance can thereby be adapted to the active region of activation functions

Adding batch norm to a network

$$\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\gamma + \beta$$

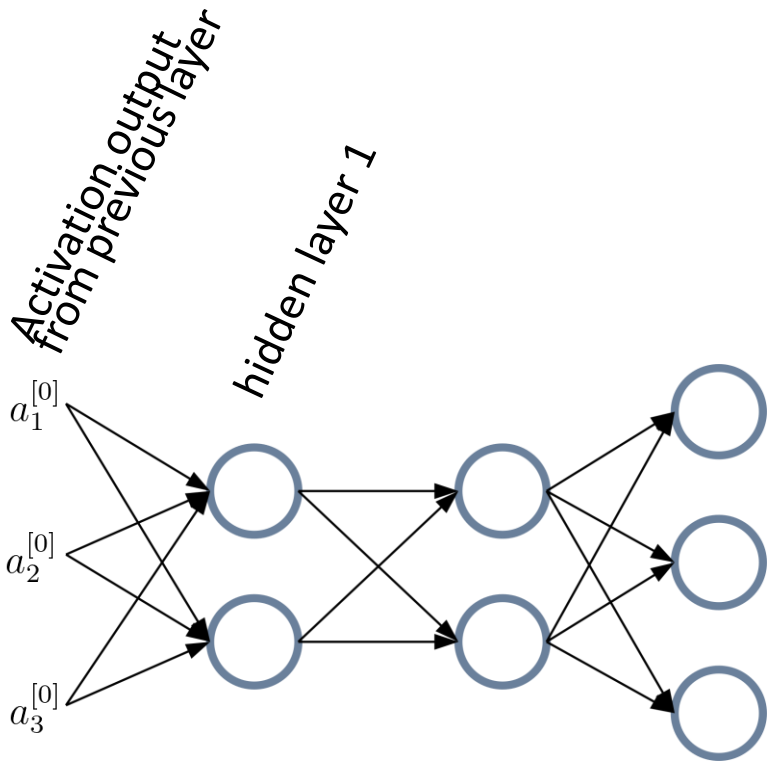
Let's draw the process chain for a single hidden layer

$$a^{[0]} \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow{\gamma^{[1]}, \beta^{[1]}} \tilde{z}^{[1]} \xrightarrow{f(\tilde{z})} a^{[1]}$$

Linear operation

batch norm

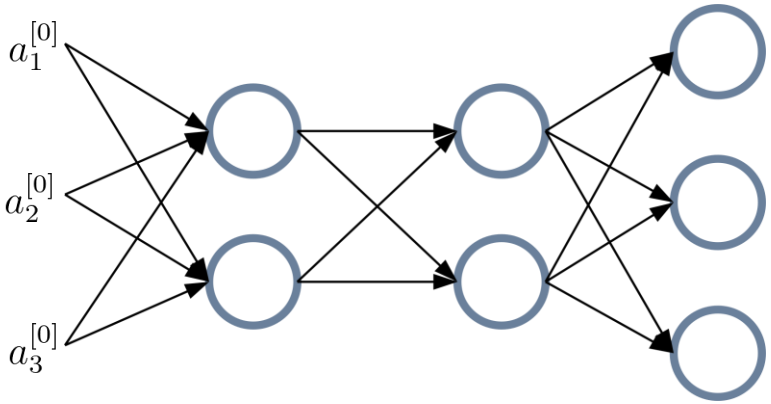
Non-linear activation (Sigmoid, ReLu, …)



NB: Batch norm removes the mean, so there is no point in having the bias *b*

Intuition: Why does Batch normalization work?

- The job of a layer is to map its input to an output value.
- When updating previous layers in the network, the input to deeper layers changes.
  - Batch norm fixes the input range, which makes the weights more robust to changes in previous layers or changes to input distribution.



Batch normalization in PyTorch

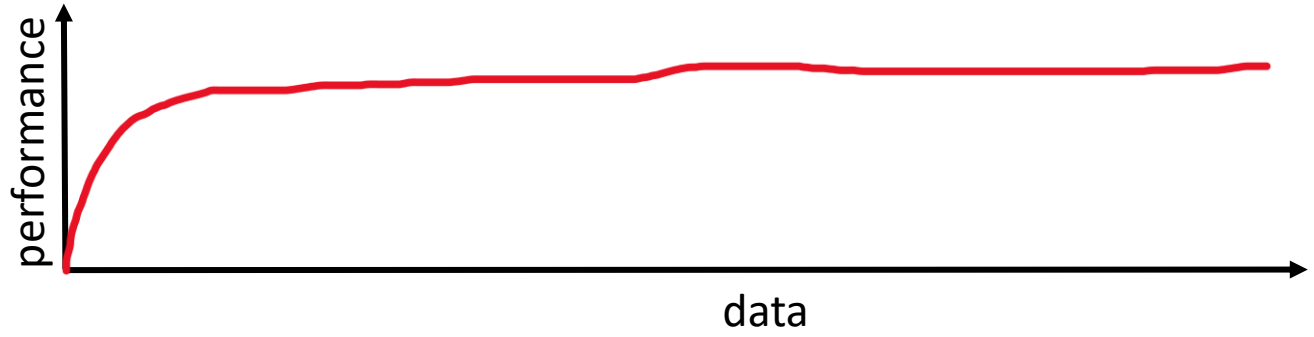
Implementing batch normalization in PyTorch requires just a single line of code for each layer you want to apply batch norm to.

```
torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)
```

Data as regularization

General rules-of-thumb

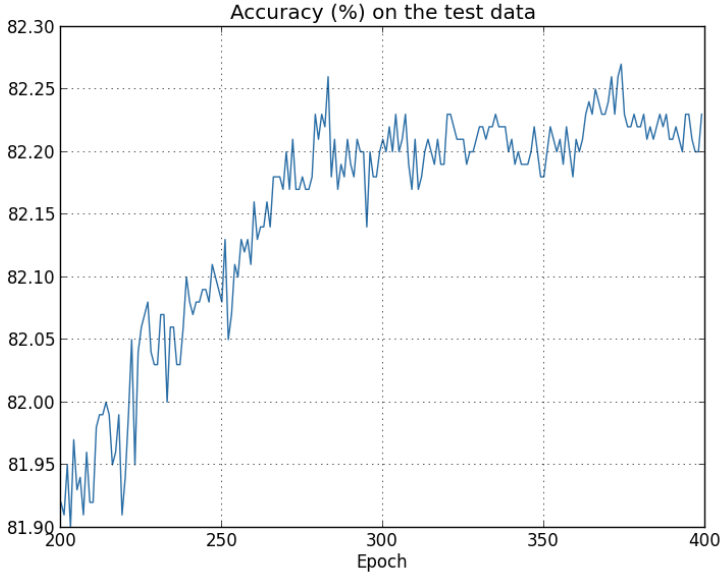
- If you have a big gap between the performance on the **validation data** and the **training data**, I.e., variance problem (maybe overfitting), we might need some regularization.
  - We could add more data as regularization to reduce the gap between the data sets.
- If the performance of the model is far from human performance, it might be a problem with the architecture and not the amount of data



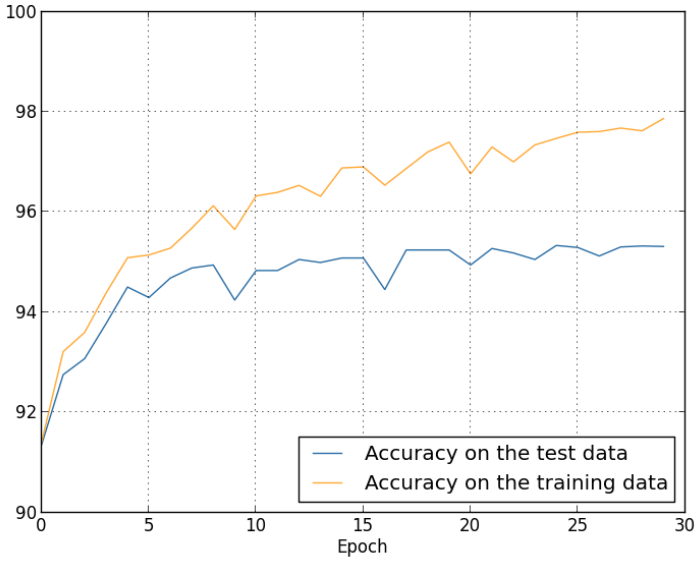
Data as regularization

Remember the previous slide where a network was trained on 1000 MNIST samples? If we add more training data, we bring the test and training accuracy much closer. Overfitting is still going on, but it's been greatly reduced.

Training on 10 000 images



Training on 50 000 images



Images from "Neural Networks and Deep Learning" by Michael Nielsen

If more data is better, why
don't we just add more
more data???

If more data is better, why don't we just add more more data???

It is often easy to get more data, but hard to get annotated data

How can we get more data from our existing data?

WE NEED MORE DATA!

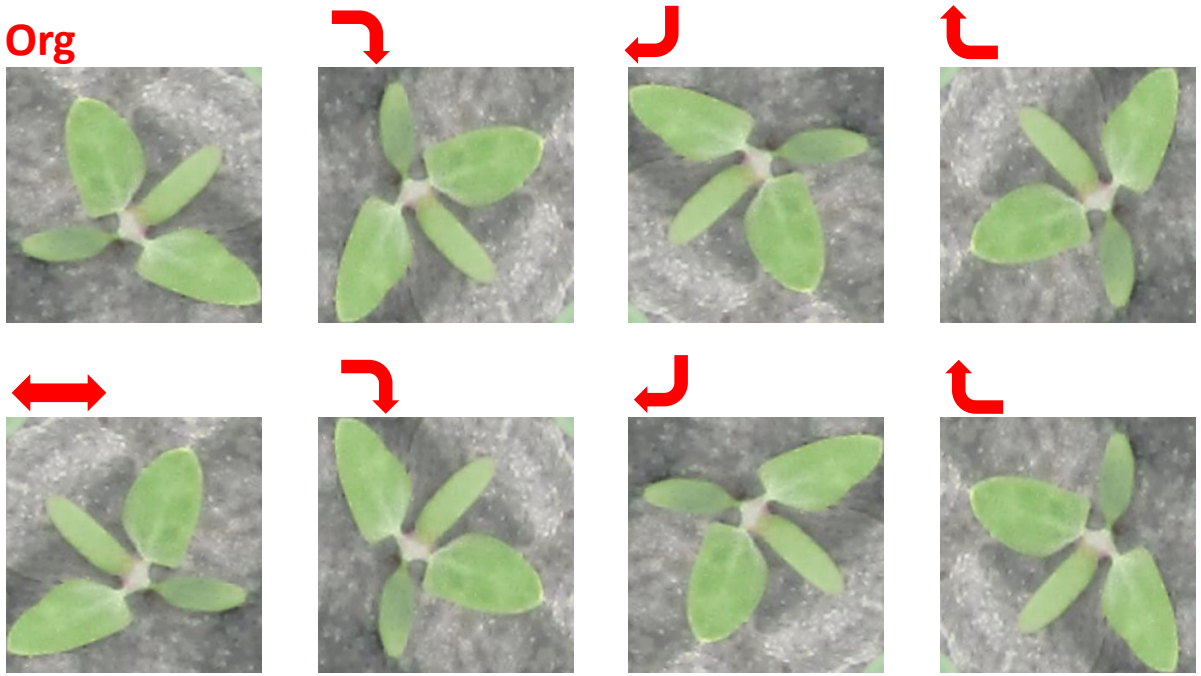
Can we in some easy way induce extra variation into our training set to help regulating the training process?

Yes! This is called data augmentation

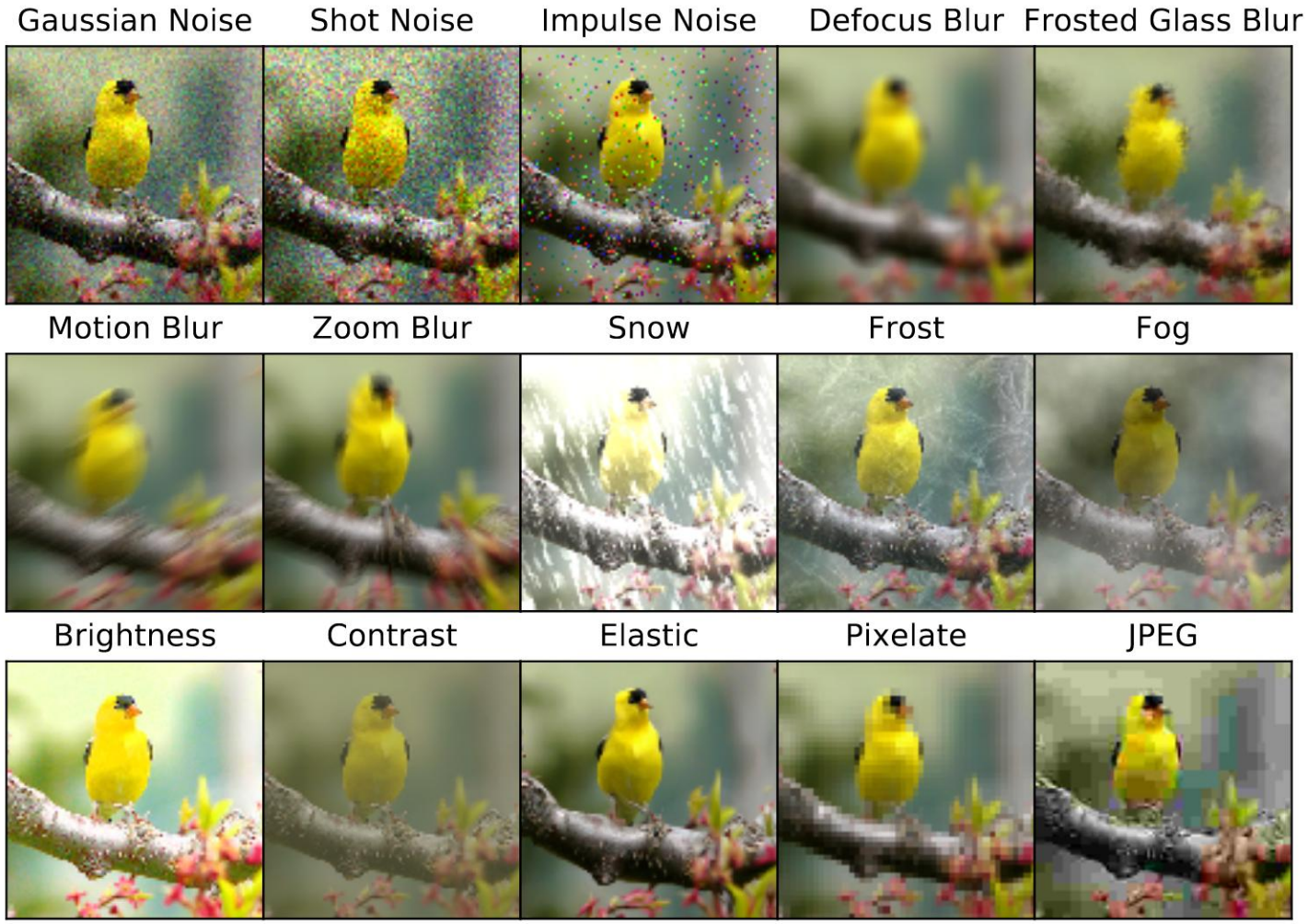
Image data augmentation

- Remember that CNNs are not rotation invariant?
- In many image-classification tasks we can double our training set by vertical flipping the images
- If images are taken top-down, we can eightfold the amount of data by flipping and 90-degree rotations.
- Cheap, since we are just switching indexes

left-right flipping

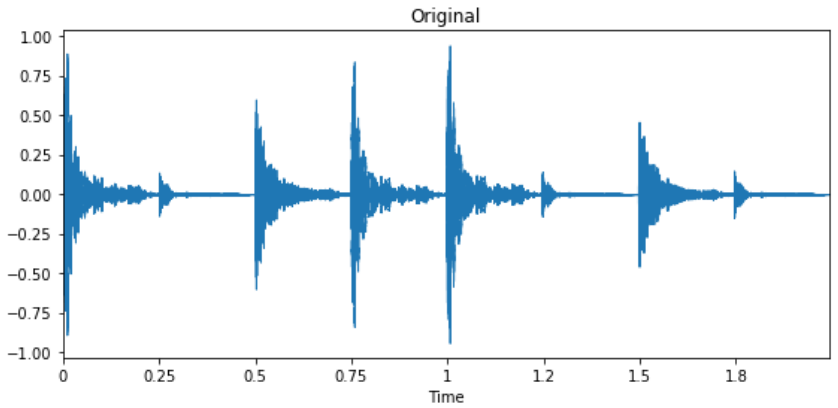


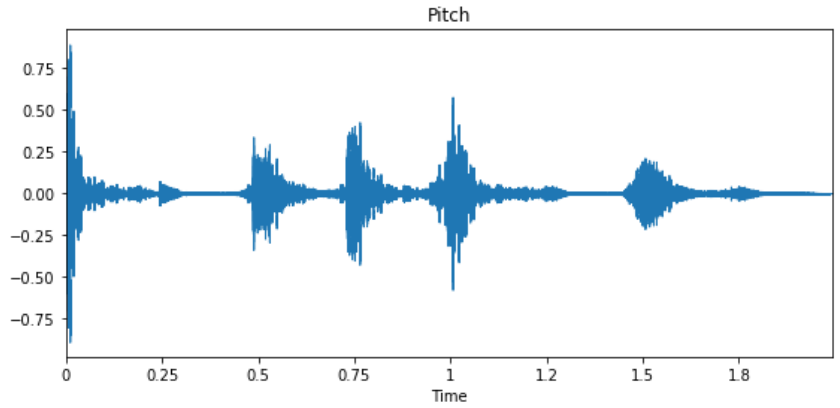
Various types of image augmentation



Data augmentation for audio

- Noise injection
- Shifting Time
- Changing Pitch
- Changing Speed
- …





Can you add too much data augmentation?

I believe you can.

If you distort a characteristic feature of a given class too much, the network will learn to not pay attention to that feature.

Usually, green is characteristic for plants, but shifting color too much will teach the network that green is not a valid indicator.

Know your data and make it look realistic and reflect the actual context.

PyTorch transforms

- In pytorch you can specify a series of preprocessing steps for your data.
- https://pytorch.org/vision/stable/transforms.html

- Remember to only distort your training data, not your test data

```
transformTrain = transforms.Compose(
   [transforms.ToPILImage(),
    transforms.Resize((224,224)),
    torchvision.transforms.RandomHorizontalFlip(p=0.5),
    torchvision.transforms.RandomVerticalFlip(p=0.5),
    … <your own transforms>
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

Image augmentation libraries

Albumentations

https://albumentations.ai/

https://albumentations-demo.herokuapp.com/

Imgaug

https://imgaug.readthedocs.io/en/latest/

https://github.com/aleju/imgaug

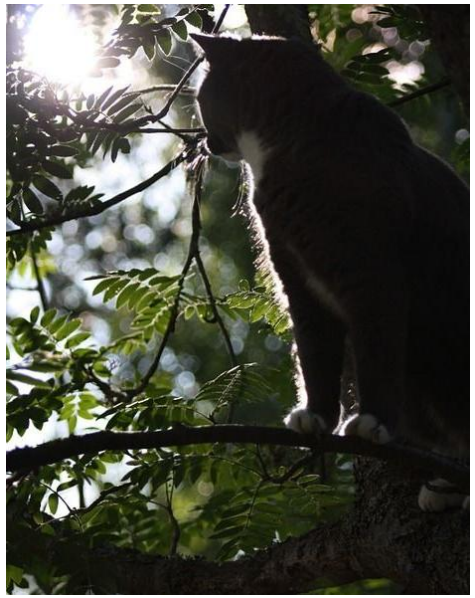
Augmentor

https://github.com/mdbloice/Augmentor

Challenges: Viewpoint variation

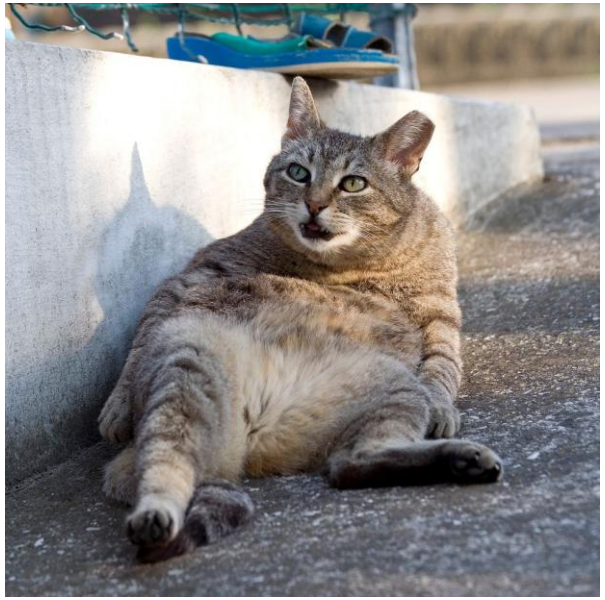
This image is CC0 1.0 public domain

Challenges: Illumination



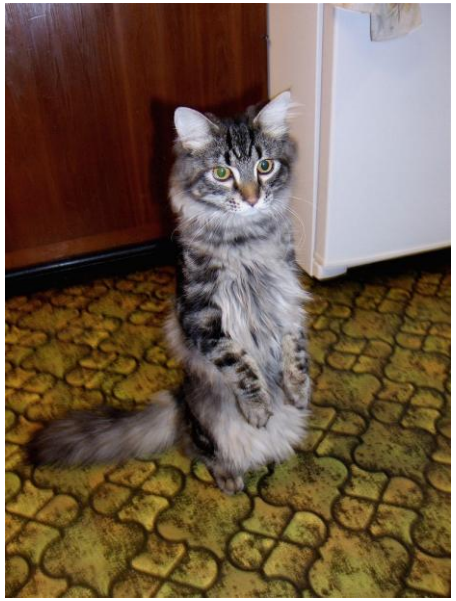
These images are CC0 1.0 public domain

Challenges: Deformation

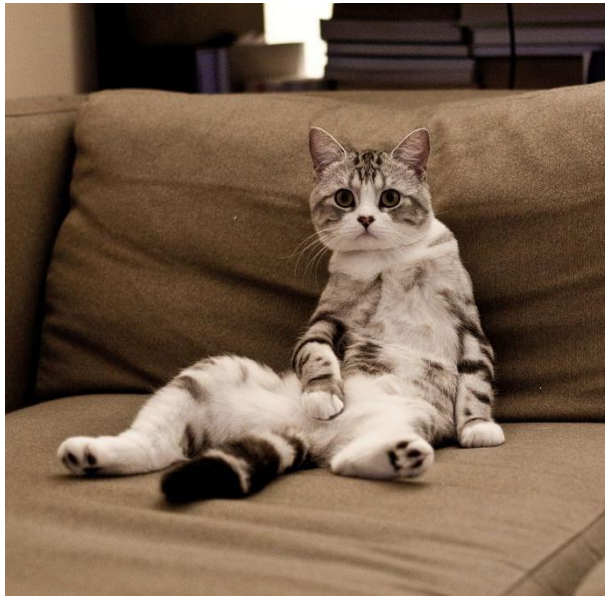


This image by Umberto Salvagnin is licensed under CC-BY 2.0

This image by Umberto Salvagnin is licensed under CC-BY 2.0



This image by sare bear is licensed under CC-BY 2.0



This image by Tom Thai is licensed under CC-BY 2.0

Slide from: CS2031n, Fei-Fei Li & Justin Johnson & Serena Yeung

Challenges: Occlusion

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

This image by jonsson is licensed under CC-BY 2.0

Slide from: CS2031n, Fei-Fei Li & Justin Johnson & Serena Yeung

Challenges: Background Clutter

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

Slide from: CS2031n, Fei-Fei Li & Justin Johnson & Serena Yeung

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Slide from: CS2031n, Fei-Fei Li & Justin Johnson & Serena Yeung

Exercise

- Take the code that you have in connection to last week, then try the following:

(i) Set up an early stopping criteria – maybe stop training when the validation loss does not decrease for 10 consecutive epochs.

(ii) Try different optimizers, see if you notice anything in their training progress (keep early stopping criteria).

(iii) Can you make a network that overfits?

(iv) Apply regularization to see if you can prevent overfitting (such as L2 norm, batch normalization, dropout)

(v) See if data augmentation helps [optional]