Deep Neural Networks
And Where to Find Them

Lecture 4

Artem Korenev, Nikita Gryaznov

Recap

Recap – Types of Problems and Losses

Most frequent problems:

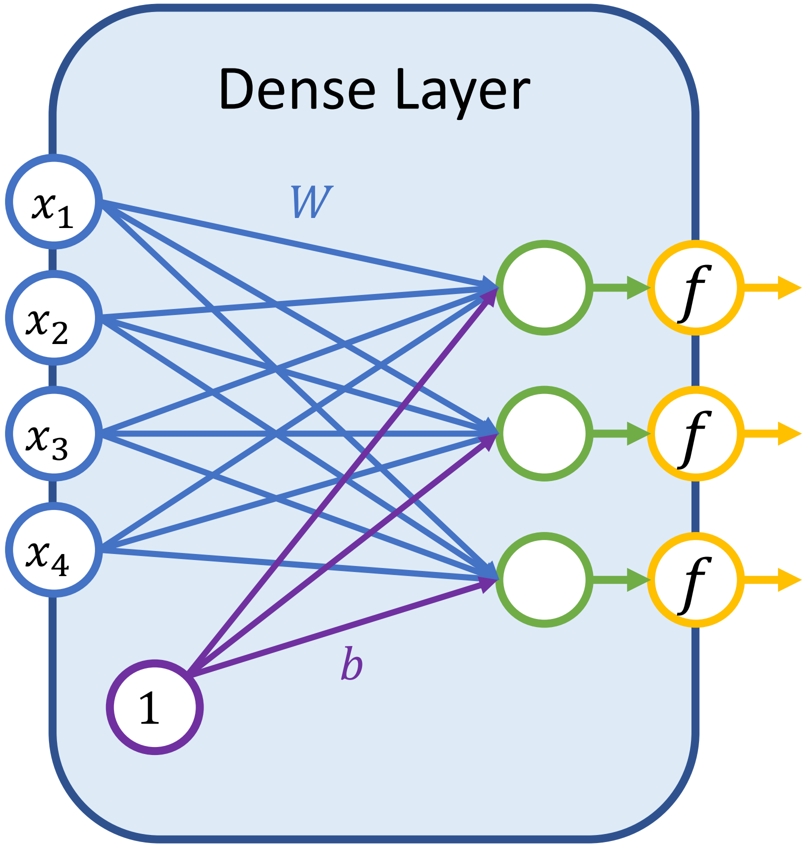
- Classification (predicting label(s) across the predefined set)

  Losses:
  - Binary Cross Entropy (2 classes) or Categorical Cross Entropy (>2 classes)
  - (rarely) Hinge Loss

- Regression (predicting real value without predefined set of outcomes)

  Losses:
  - Mean Squared Error (L2 Loss)
  - Mean Absolute Error (L1 Loss)

Recap – Dense Layer



x_1 Layer input

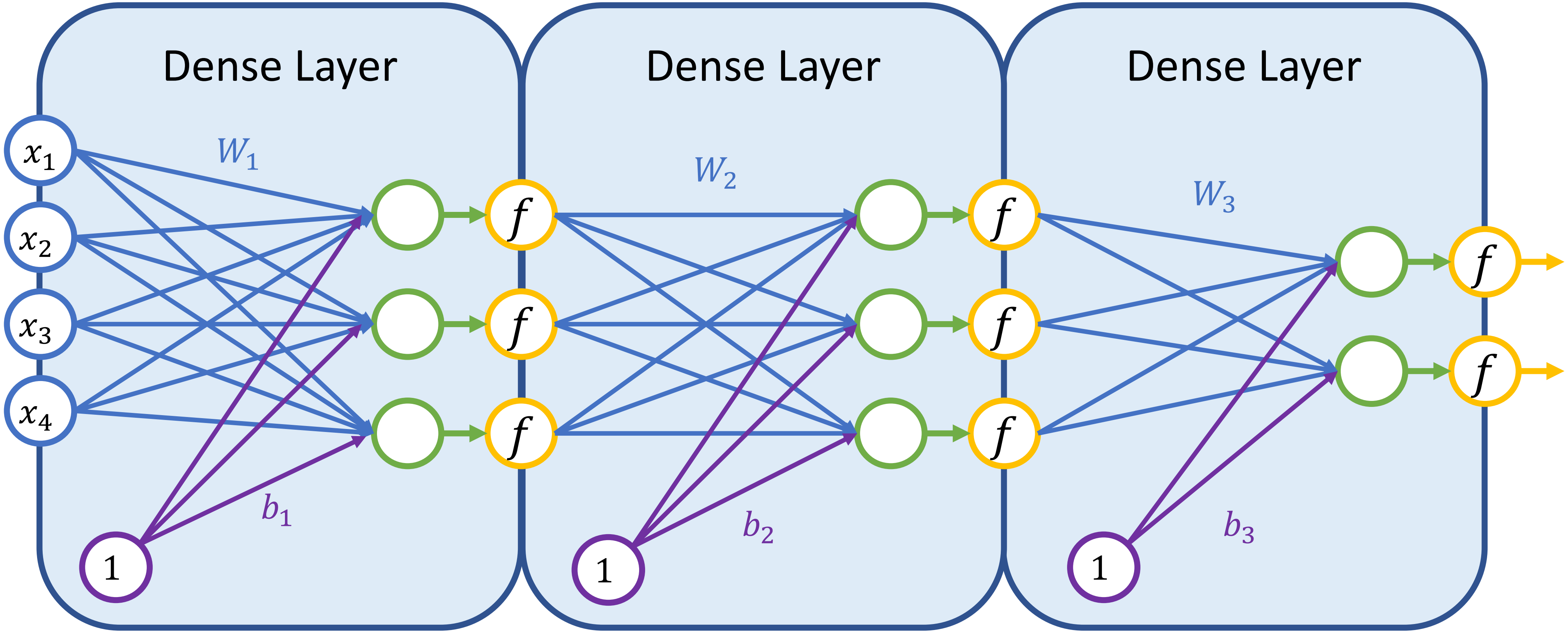
Neuron Output
$(W_{i1}x_1 + \cdots + W_{i2}x_2 + b_i)$

f Activation function
(Non-linearity)
(Neuron activation)

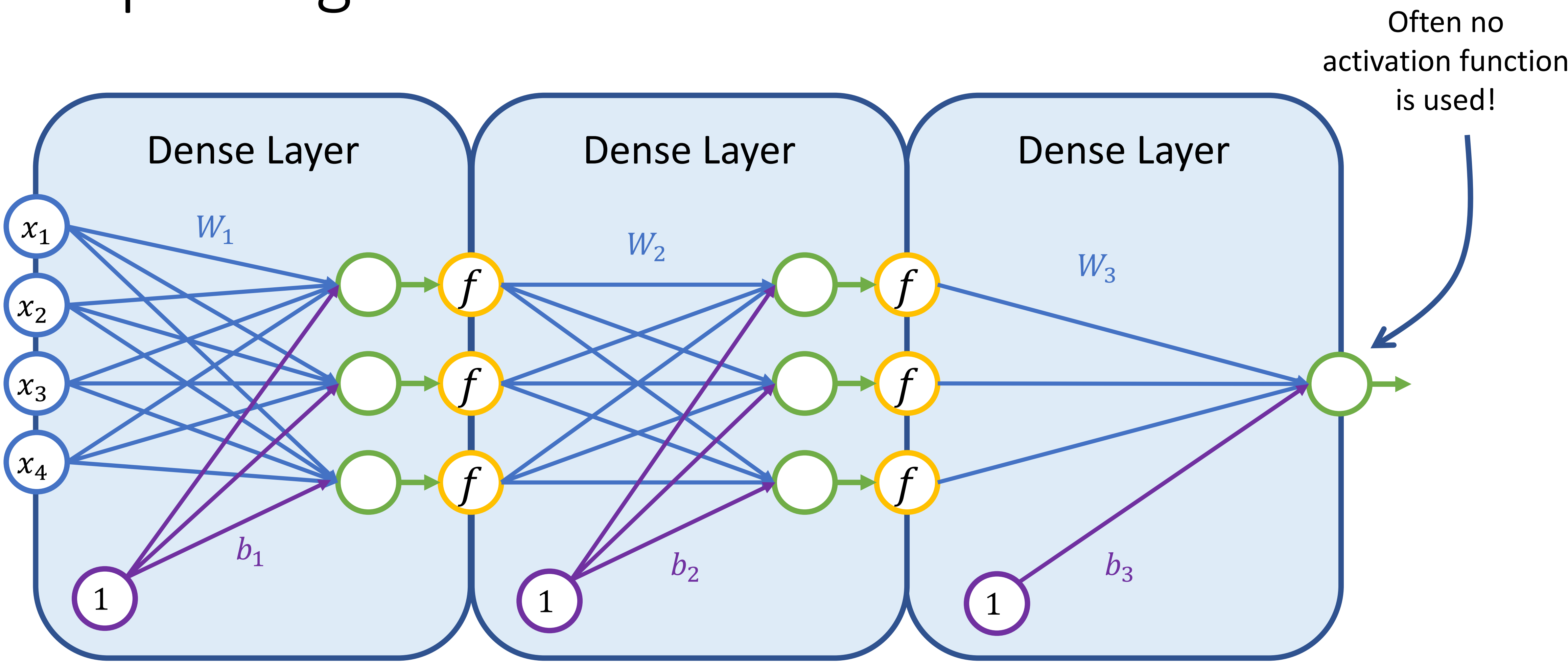
Layer weight
(trainable parameter)

Layer bias
(trainable parameter)

Recap – Multi Layer Neural Network

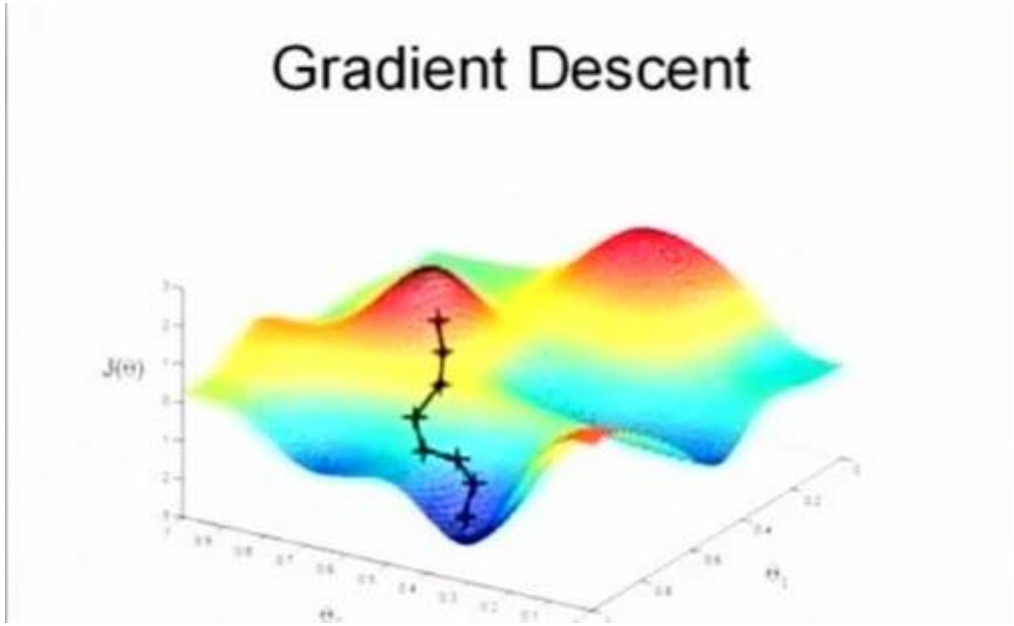


Recap – Regression Problem



Recap - Optimization

- *Backpropagation* using *chain rule* allows us to compute gradients for all parameters of deep networks
- We use *stochastic gradient descent* to optimize the network
- Stochastic means we use small batches to update the model instead of the whole dataset



Recap – Quick Points

- Use ReLU activation function as your primary choice
- Carefully search for a good *learning rate*
- Track down useful *metrics* for your problem

Regularization

Regularization

- Is a set of tools for reduction of overfitting
- Restricting your model to continue learning the same stuff it has already learnt

L2 Weight Regularization

- We add an additional term to our loss function

$$L_{total}(y, f(x), w) = L(y, f(x)) + \frac{1}{2}\lambda\|w\|_2^2 = L(y, f(x)) + \frac{1}{2}\lambda\sum_{i=1}^{n_w} w_i^2$$

- Restricting weights to be more uniform and less spiky

Sum across all
weights except
biases

L1 Weight Regularization

- We add an additional term to our loss function

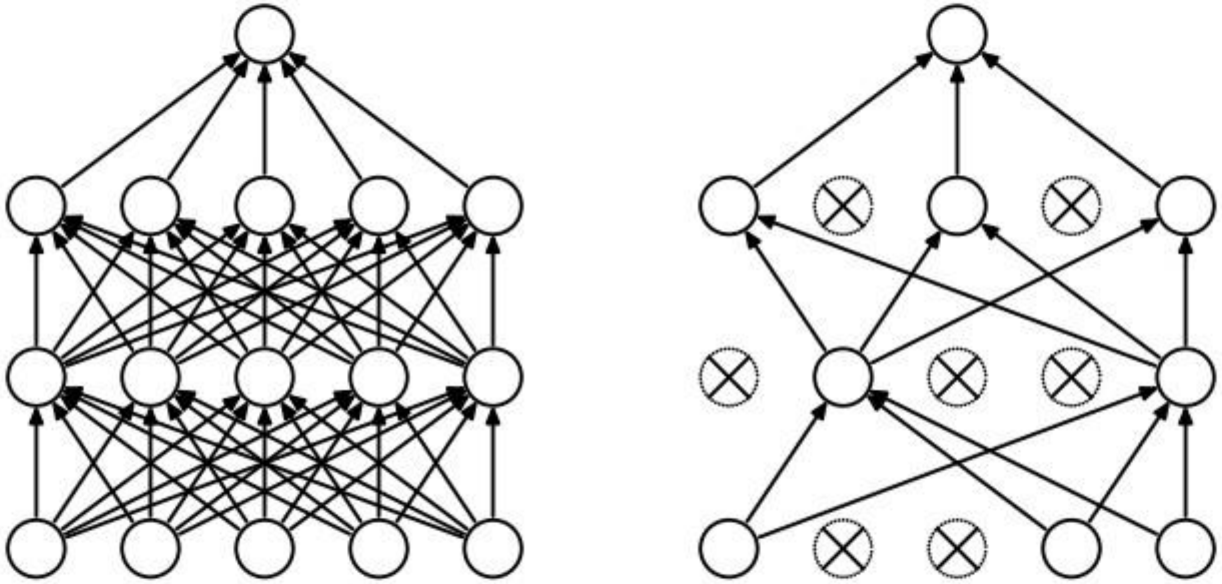
$$L_{total}(y, f(x), w) = L(y, f(x)) + \lambda \|w\|_1 = L(y, f(x)) + \lambda \sum_{i=1}^{n_w} |w_i|$$

- Restricting weights so weight matrices tend to be sparse
- Expected to work worse than L2 Weight Regularization

Sum across all
weights except
biases

Dropout

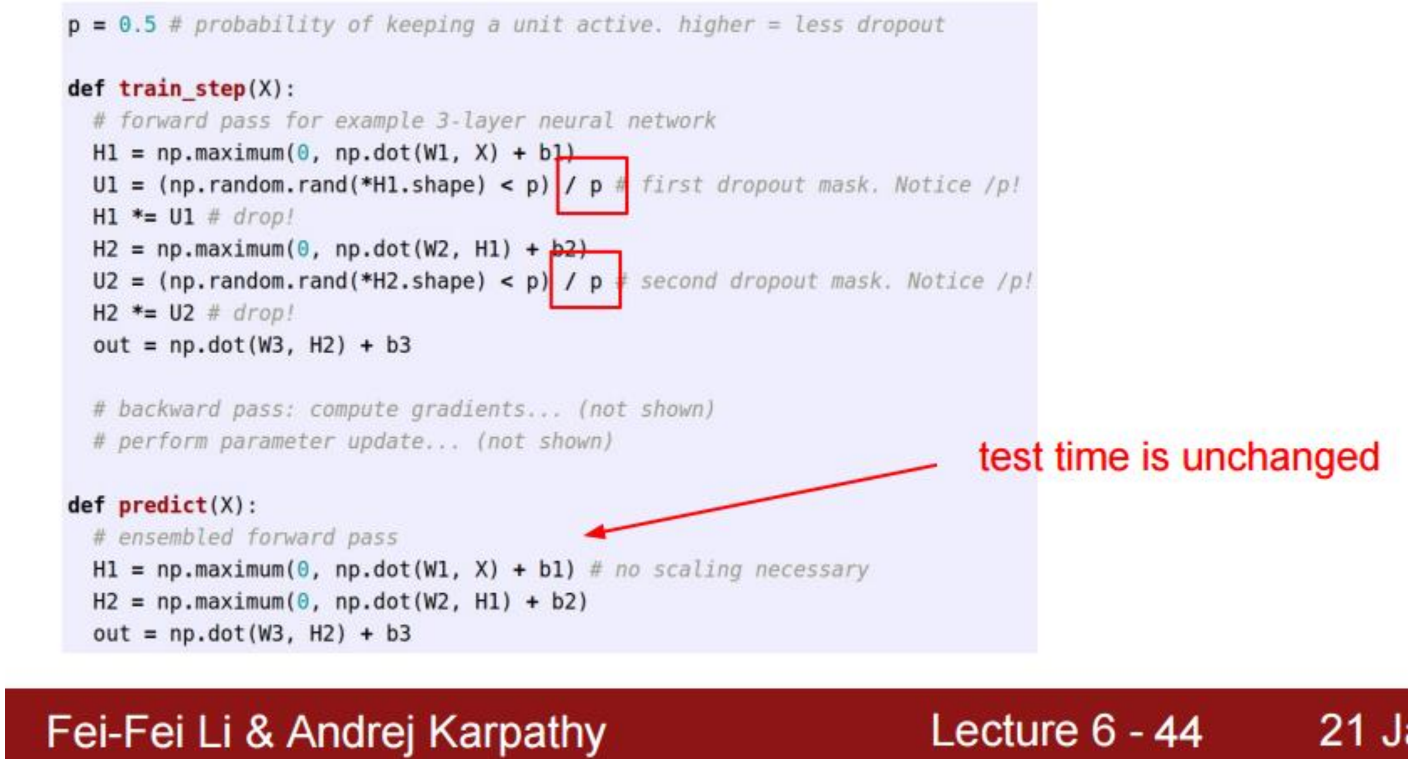
- Switching off random neurons of the layer with the given probability



(a) Standard Neural Net   (b) After applying dropout.

Dropout

- We train with dropout and switch it off outside training
- For layers with dropout with probability p we scale outputs with $\frac{1}{p}$
- We switch off scaling outside training loop



```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Batch Normalization

m - Batch Size

x - One output from the layer

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Output of Batch Normalization

Trained parameters via backpropagation

Batch Normalization

- Enforces (mostly) unit Gaussian outputs from the neuron
- Extremely powerful technique
- Decreases training time
- Allows using bigger learning rates
- Since batch mean and variance can be noisy it regularizes the network a little bit
- A rule of thumb: Dense -> Batch Normalization -> Activation

```
x = Dense(128)(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
```

Batch Normalization Outside Training

Not computed from a batch but stored during training as training dataset mean and variance (using exponential moving average)

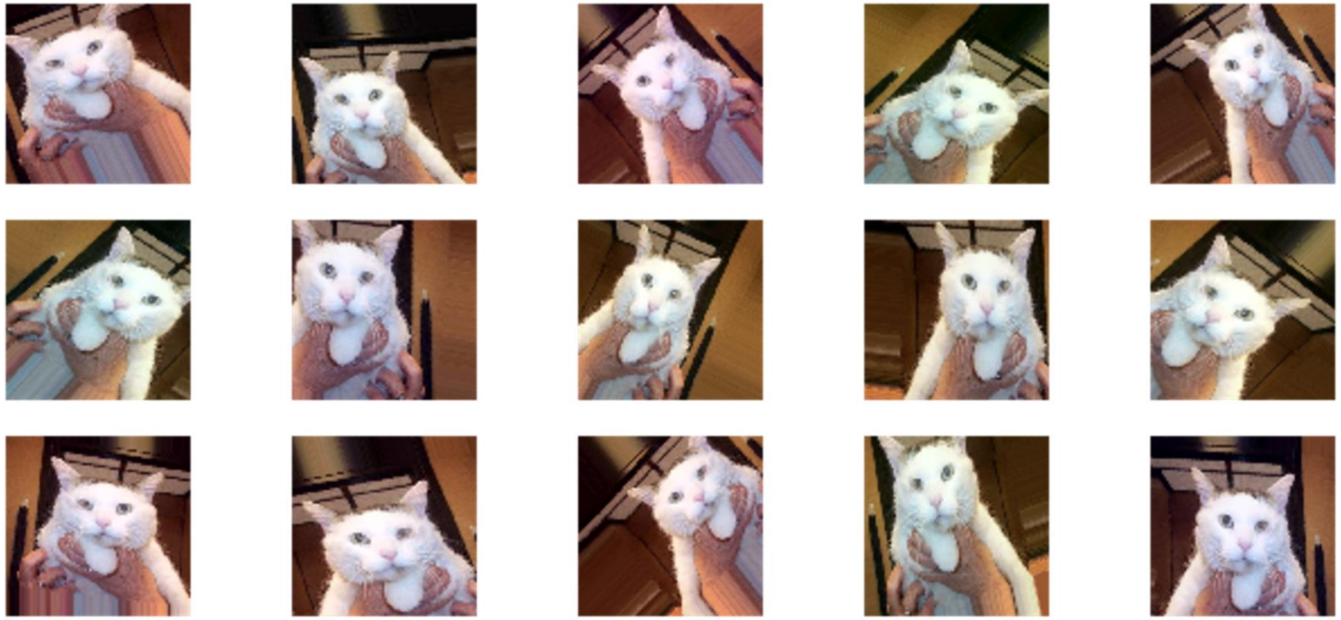
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

Freezing these trained parameters after training

Data Augmentation

- Artificially add more data (so NN can learn something new instead of overfitting to the same data)
- For images: rotating, scaling, flipping, cropping, etc.
- Very specific to your task



Useful Tips for Regularization

- Dropout is a powerful technique but makes training longer
- Batch Normalization helps a lot but use it before non-linearity
- For bigger NNs use L2 Weight regularization
- If you have a way to augment your data – go for it

Optimizers

Parameters Optimization

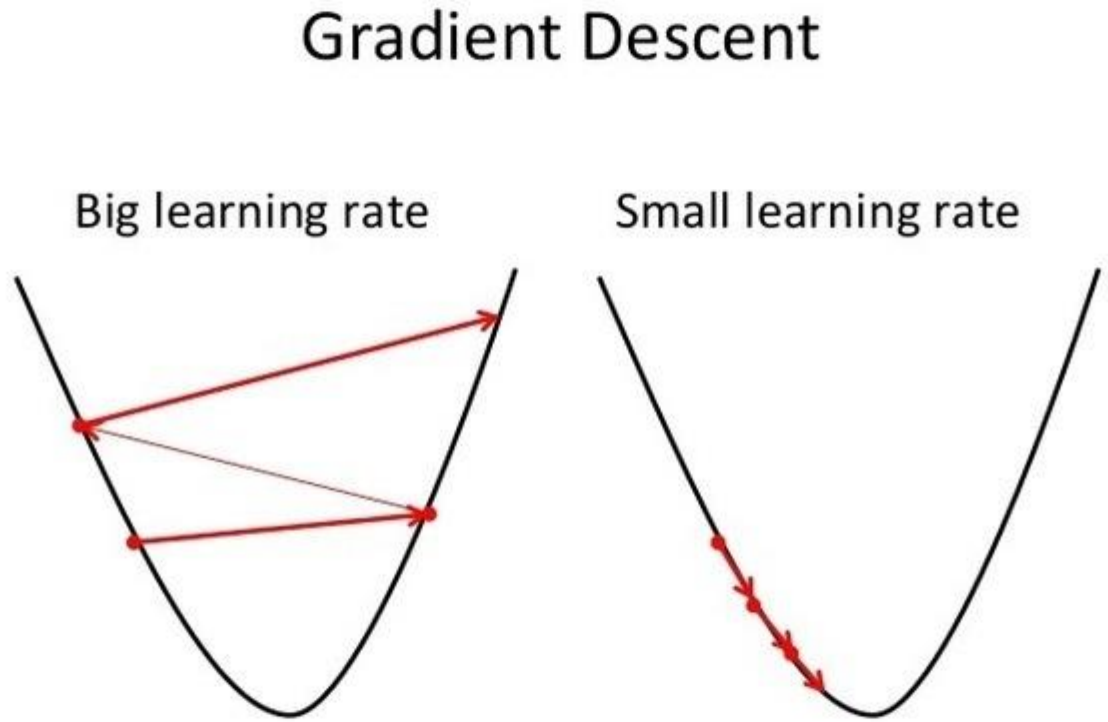
- To train neural networks we need to adjust parameters
- Optimization to the rescue
- We already know how to compute gradients, but how to use them?

Vanilla gradient descent

Most basic method

$$w_{t+1} = w_t - \alpha \nabla_w L(w_t)$$

- α is manually chosen parameter called **learning rate**



Gradient descent types

- All data – 1 update:

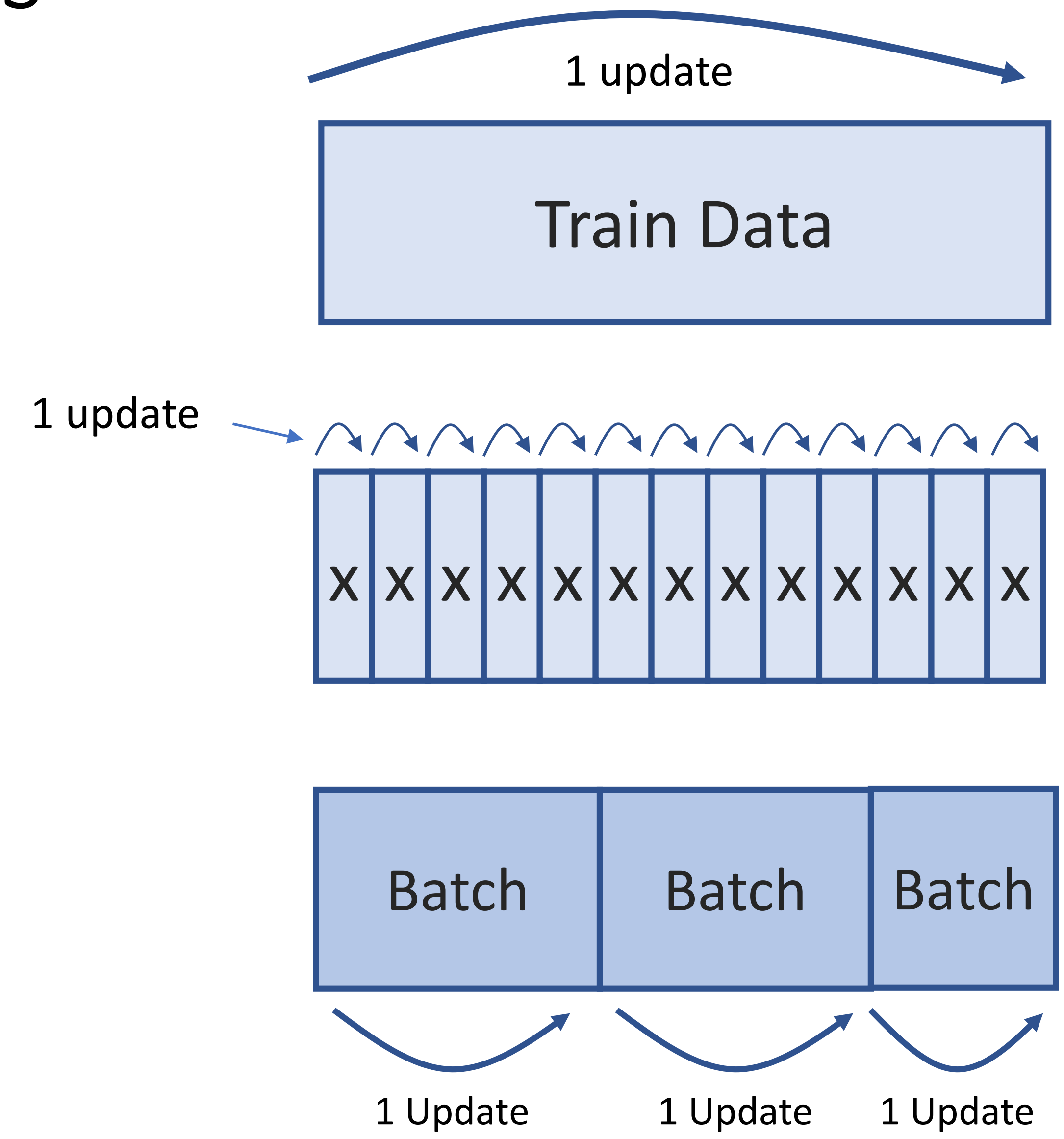
gradient descent

- 1 sample – 1 update:

stochastic gradient descent

- Batch of samples – 1 update:

mini-batch gradient descent

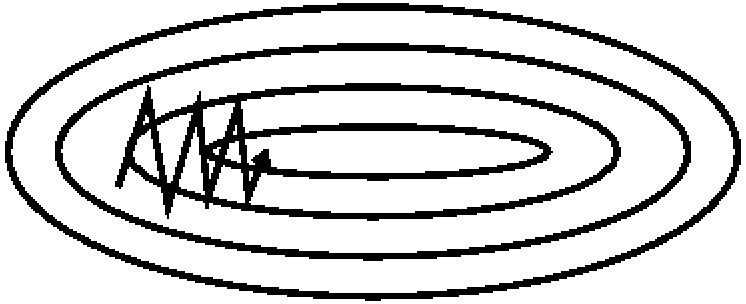


Problems at this point

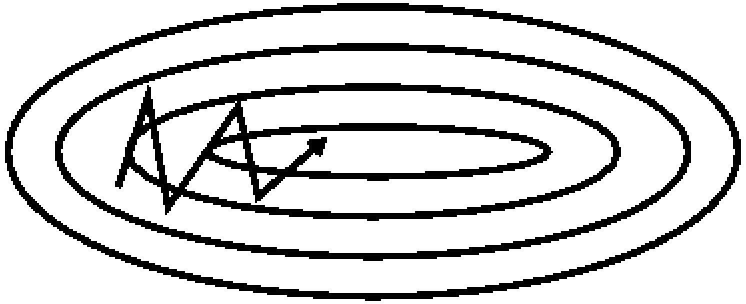
- How to choose learning rate?
- Maybe change it during time? But how?
- How to avoid suboptimal local minima and saddle points?
- How to make it faster?

Momentum

- Helps to dump oscillations when surface curves much more steeply in one dimension than in another



Without momentum



With momentum

$$v_{\mathrm{t}} = \gamma v_{t-1} + \alpha \, \nabla_w L(w_t)$$
$$w_{t+1} = w_t - v_t$$

γ is usually chosen to be around 0.9

- Ball analogy: ball accumulates momentum as it rolls downhill, becoming faster and faster on the way

AdaGrad

- Performs larger updates for infrequent parameters and smaller updates for frequent one
- Eliminates the need to tune the learning rate
- Each parameter has its own learning rate

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\varepsilon + \sum_{k=1}^{t-1} g_k^2}} g_t$$

- Monotonically decreasing
- Need to store all previous gradients
- Still have to choose initial learning rate

RMSProp

- Solves problem with monotonically decreasing AdaGrad using exponential smoothing

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{\varepsilon + E[g^2]_t}} g_t$$

- Good in practice and is used much
- Still have to choose initial learning rate (solved by AdaDelta)

What’s next?

There are plenty more similar optimization methods:

- Nesterov accelerated gradient [advanced Momentum]
- AdaDelta [advanced RMSProp]
- Adam – use this or RMSprop if you don’t know what to choose
- AdaMax
- Nadam

To get more details read [this awesome article]

Normalization

- To make optimization faster and more robust the good idea is to normalize your data so it is zero-centered and has same scale.
- There are two ways to do it:

1. Normalization, so all features have zero mean and unit variance

$$X = \frac{X - \bar{X}}{\sqrt{Var(X)}}$$

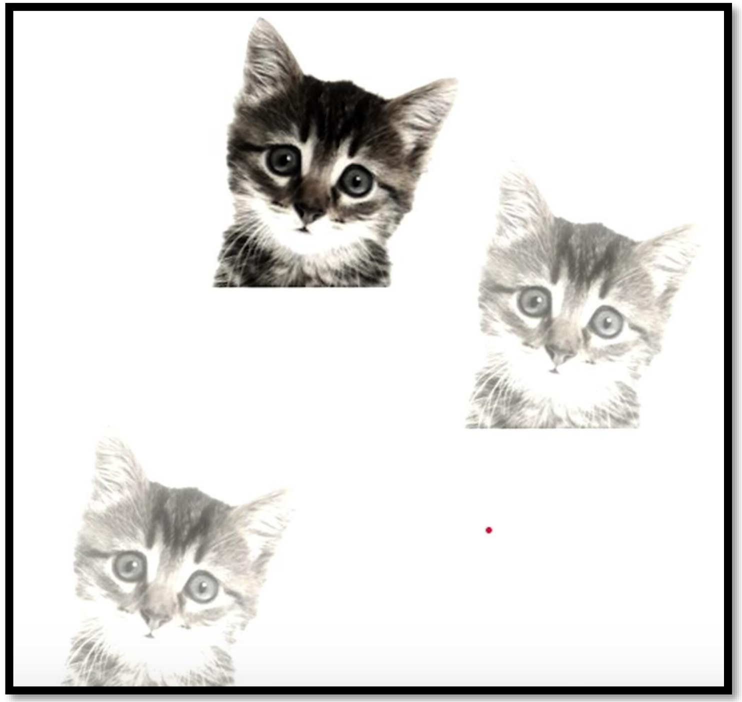
2. Normalization, so all features lie in range [-1,1]

$$X = 2\frac{X - \min(X)}{\max(X) - \min(X)} - 1$$

Intro to Convolutional NNs

Motivation

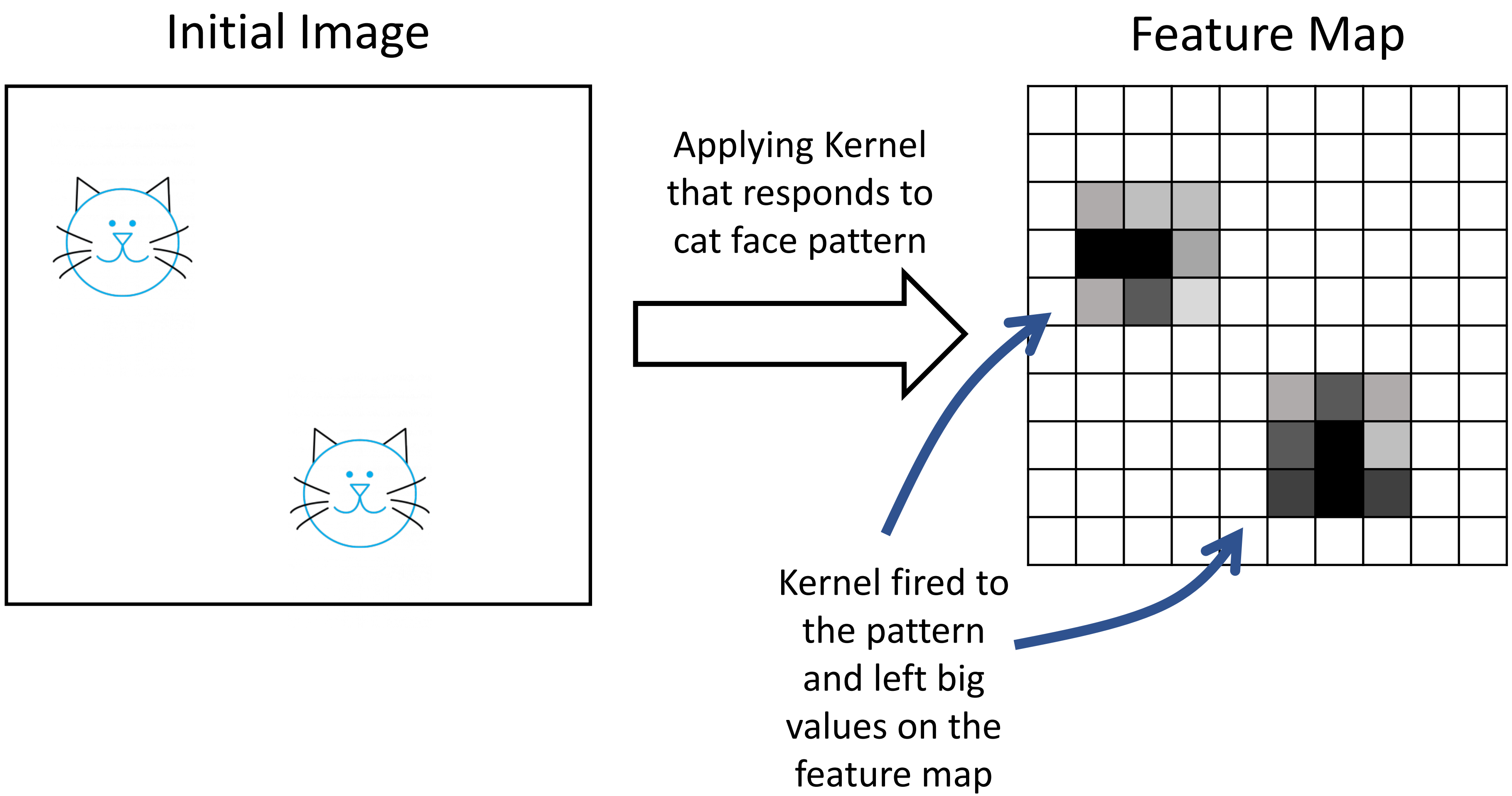
- Let's say we have many spatially independent patterns that we want to spot
- How do we learn them using Dense Layers?
- It's hard and too data consuming



Motivation

- Instead of that, we can learn small *filters* that are applied across the whole image
- These filters called *kernels* and we use *convolutions* to apply them to input images
- Consider that we have a kernel that *fires* when it sees a cat face pattern on the image
- Then applying convolution to our image with this kernel we obtain a feature map with activations corresponding to cats on the image

Convolution



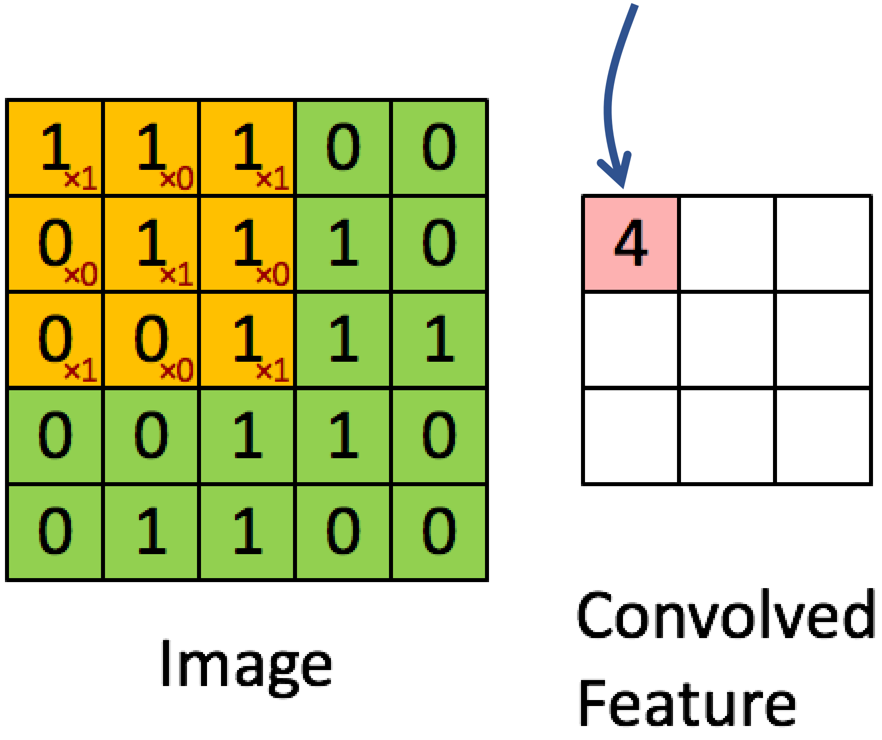
Convolution

$$\begin{matrix} 1*1+1*0+1*1 \\ + \\ 0*0+1*1+1*0 \\ + \\ 0*1+0*0+1*1 \end{matrix} = 4$$

Kernel
(i.e. filter)
has size 3x3

Kernel Values

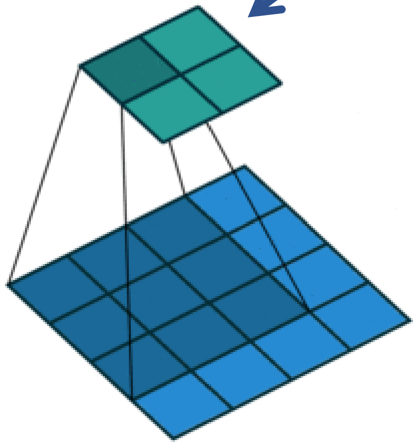
1	0	1
0	1	0
1	0	1



Convolution

Result of the convolution

Kernel
(i.e. filter)
has size 3x3



Initial Image

Convolutions with Many Channels

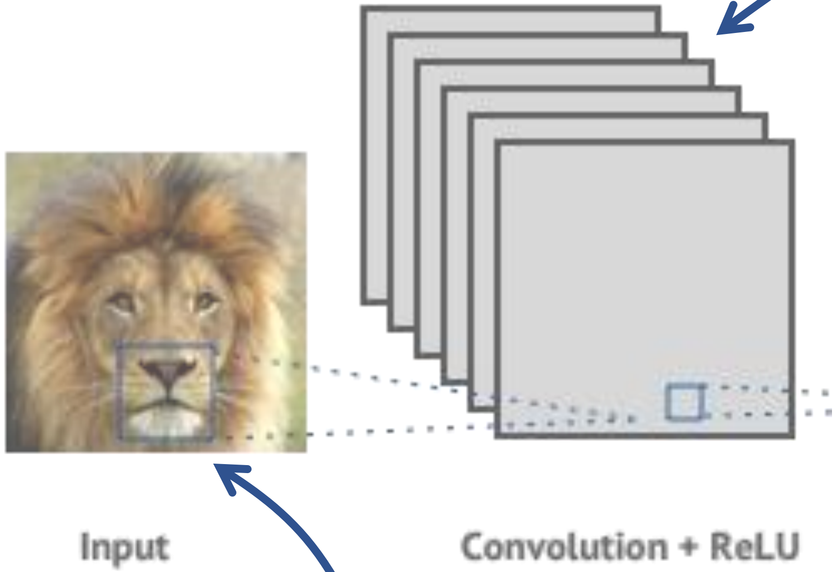
- Normally we have images with 3 color channels (Red, Green, Blue)
- Thus, we learn 3-dimensional *kernels* with sizes $Height * Width * Channels$
- We learn many *kernels* with the same sizes at once and stack their results to each other
- We learn *bias* weights for kernels as well
- After convolution we apply non-linearity as in Dense Layers

Kernel Channels

We applied 6 different kernels,
therefore we obtain 6 feature maps

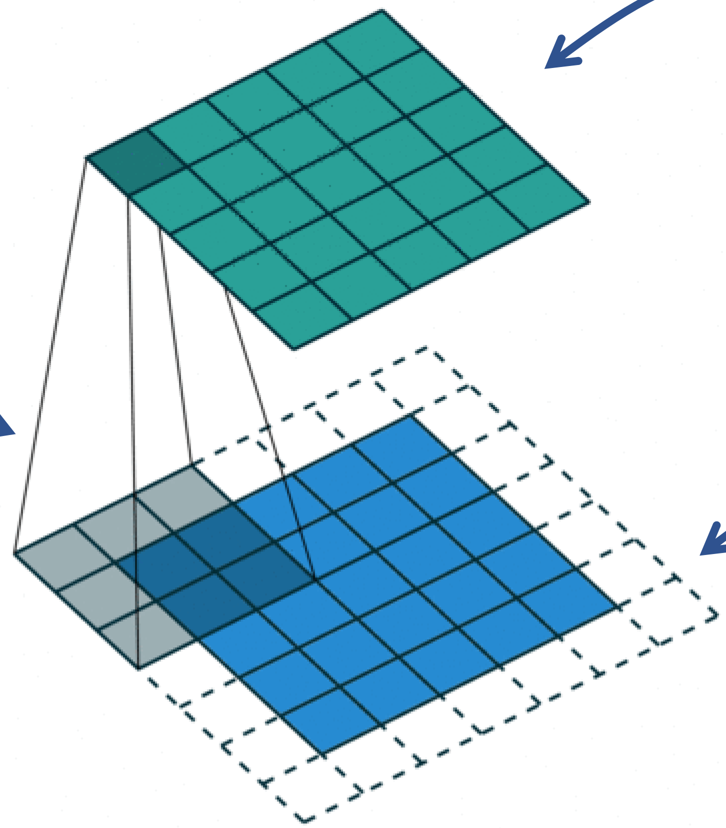
Now we can treat these 6 feature maps as a new image with 6 channels to pass it further so we can apply new HxWx6 kernels on the next layer

3 color channels for this image (RGB)



Let’s assume that this window has size of 20x20 pixels
Therefore we apply kernels of size 20x20x3 to the image

Convolution – Padding



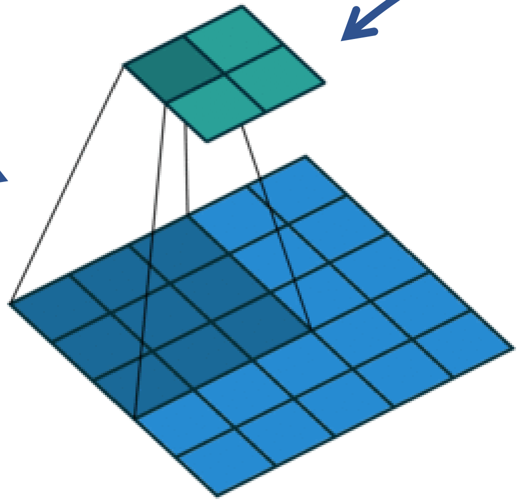
Result of the convolution has the same size as the input

For filter 3x3 padding of size 1 will result the same output

White part filled with zeros, for instance

Convolution - Strides

Stride makes ‘jump’ not by 1 pixel but by $1 + stride$ pixels



Because of these jumps output image becomes even smaller

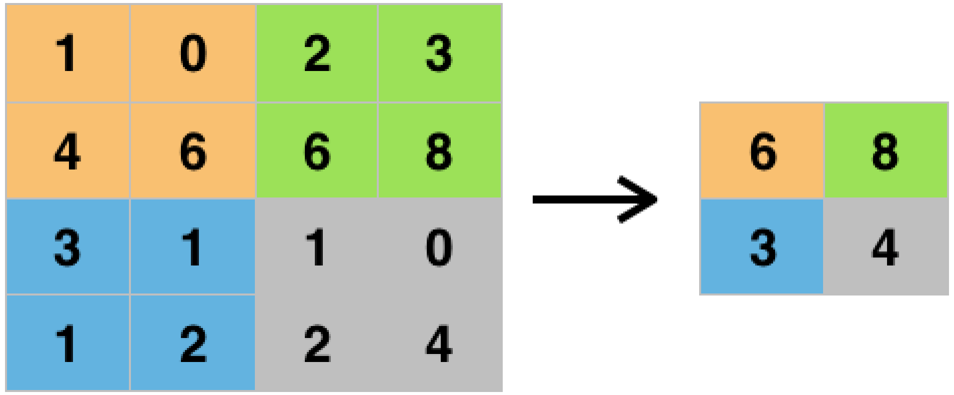
Convolution – Padding and Strides

- There is no general rule of how to use padding and strides
- Should depend on your problem
- Padding and Strides depends on amount of computations
- Therefore it also affects time for NN result computation
- For the most cases no Stride is used and Padding is either absent or such that the size of the image doesn’t change

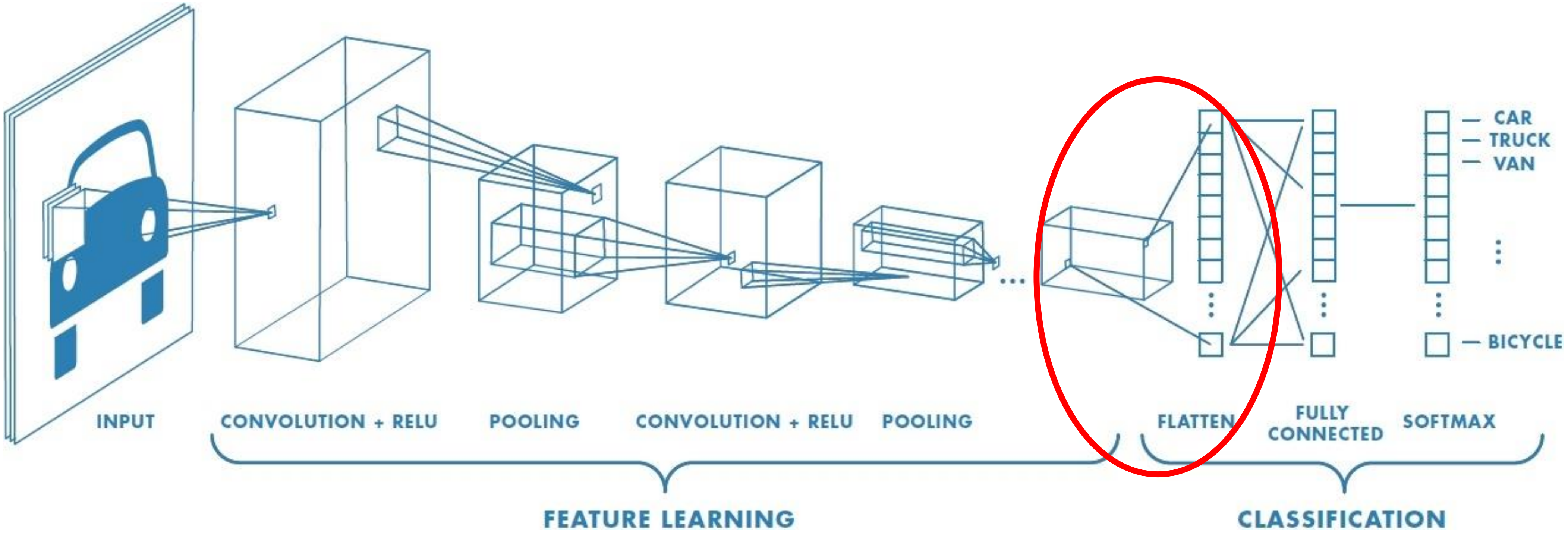
Max Pooling Layers

- Getting max value in non-overlapping windows of feature maps
- Down sampling feature maps
- Reduces overfitting
- Reduces number of data to process

Max Pooling window size is set to 2x2



Basic Structure of Deep Convolutional NN



Flattening the feature map – making one shallow vector out of the 'image'

Recap

- We learn small filters (called *kernels*) that catch useful features and pass them over
- We make many layers with such *kernels* which catch more and more complex features throughout the network
- We add *Max Pooling* layers to add more non-linearity and reduce amount of data to process by NN
- At some point we change our feature map into vector with *Flattening* and proceed as a standard NN

This is It For the Forth Lecture