Part I
Gene Regulation

- All cells in an organism contain all the organism's DNA, but we have multiple cell types (i.e. neurons, cardiomyocytes, etc.)

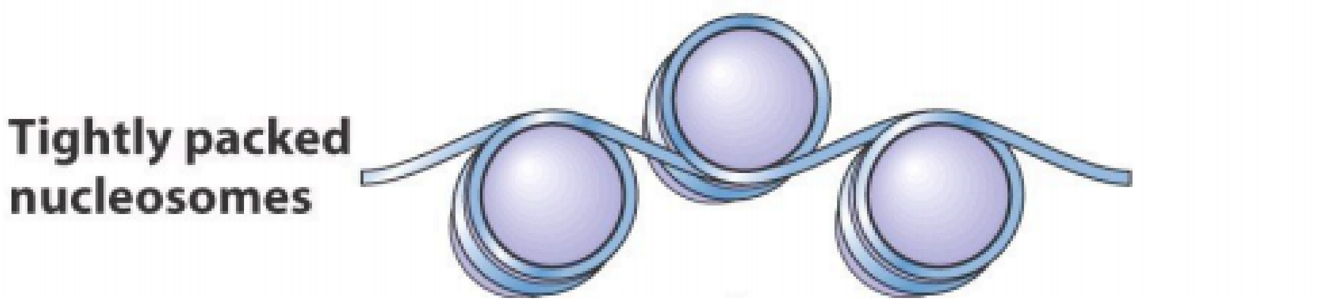
- Specific subsets of genes are turned on in different types of cells to determine cell type and function.

- There are two key players responsible for gene regulation:

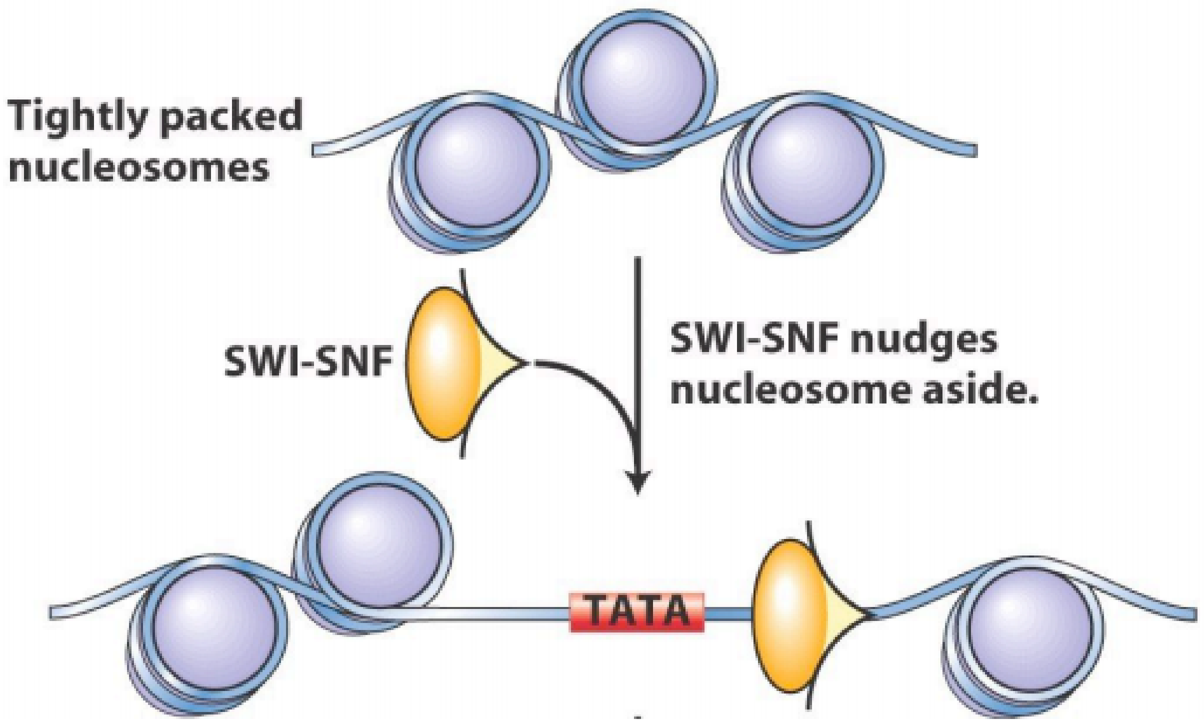
  - **Transcription factors**
    - Proteins
    - Trans-acting elements: diffuse through the cytoplasm and bind to far-away regions of DNA

  - **Motif sequences**
    - DNA sequences
    - Cis-acting elements: act at fixed position along the DNA molecule

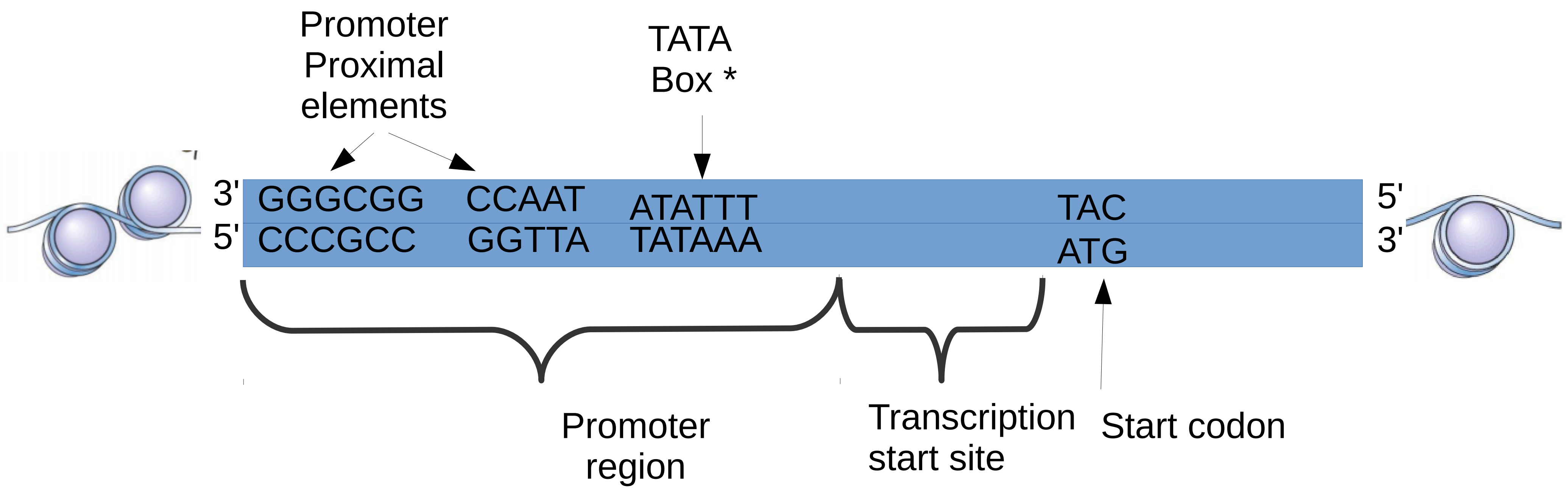
Transcription Initiation



Transcription Initiation



Transcription Initiation



- The TBP (TATA-binding protein) binds to the core TATA box region
- Transcription factors bind to the promoter proximal elements. These determine the true level of expression.
- Various combinations of core and proximal elements are found near different genes.

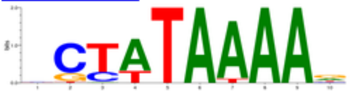
*The TATA box is an example. Not all genes have TATAAA in the promoter region, but they usu. have a motif sequence that proteins can bind to, such as CAAT box (CCAAT), GC box (GGGCGG), octamer consensus sequence (AGCTAAAT), etc.

Remember the ENCODE motifs from lecture?
TATA box is an example of such a motif.

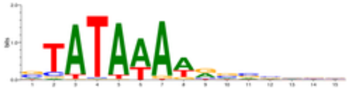
http://compbio.mit.edu/encode-motifs/

- Like other motifs, TATA has a PWM (position-weight matrix).
- Some positions are more important than others, and you observe a “taller” letter in the PWM.
- There are many “variations” of TATA, some look nothing like the canonical motif!

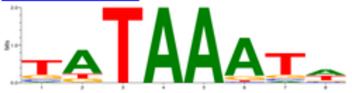
TATA_known1:



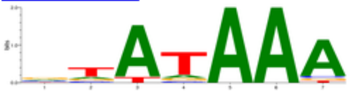
TATA_known2:



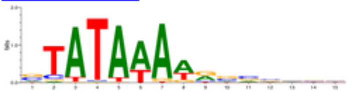
TATA_known3:



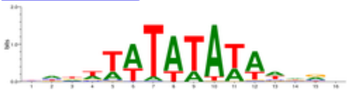
TATA_known4:



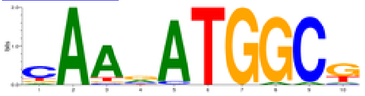
TATA_known5:



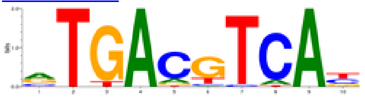
TATA_known6:



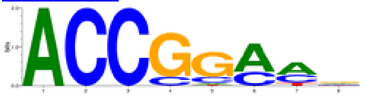
TATA_disc1:



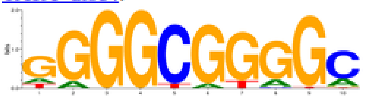
TATA_disc2:



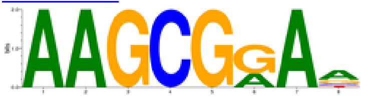
TATA_disc3:



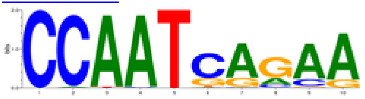
TATA_disc4:



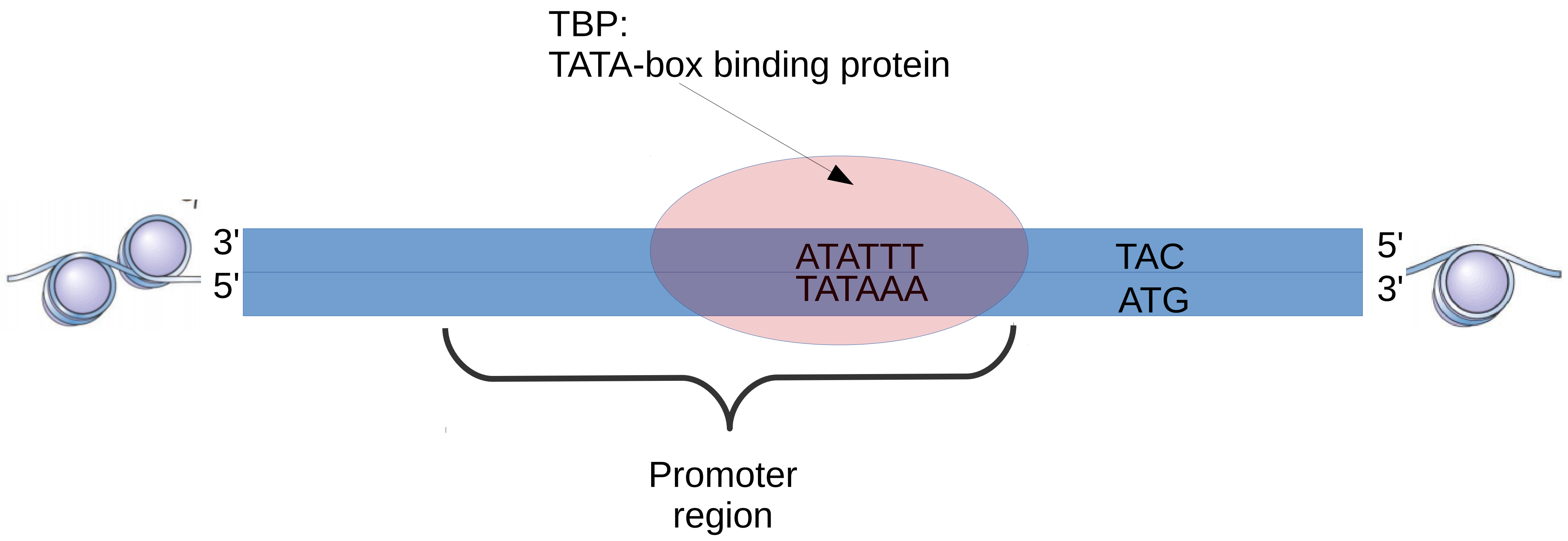
TATA_disc5:



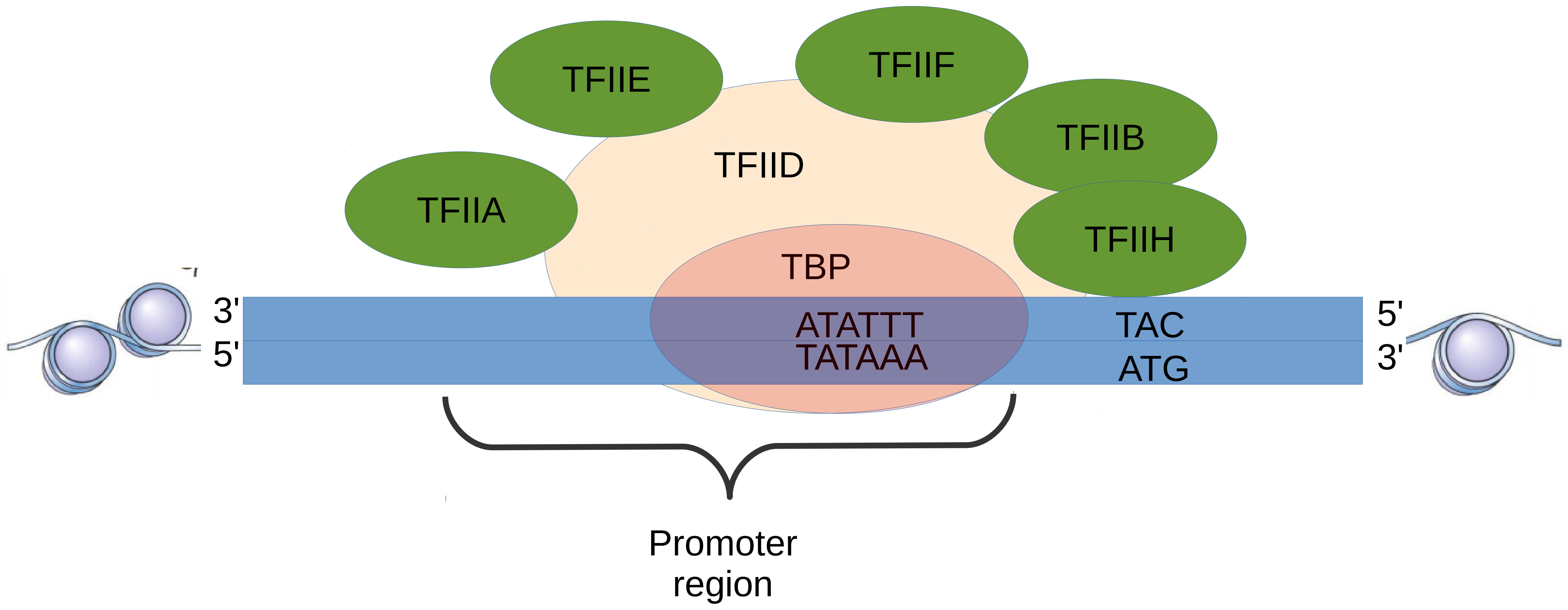
TATA_disc6:



Transcription Initiation

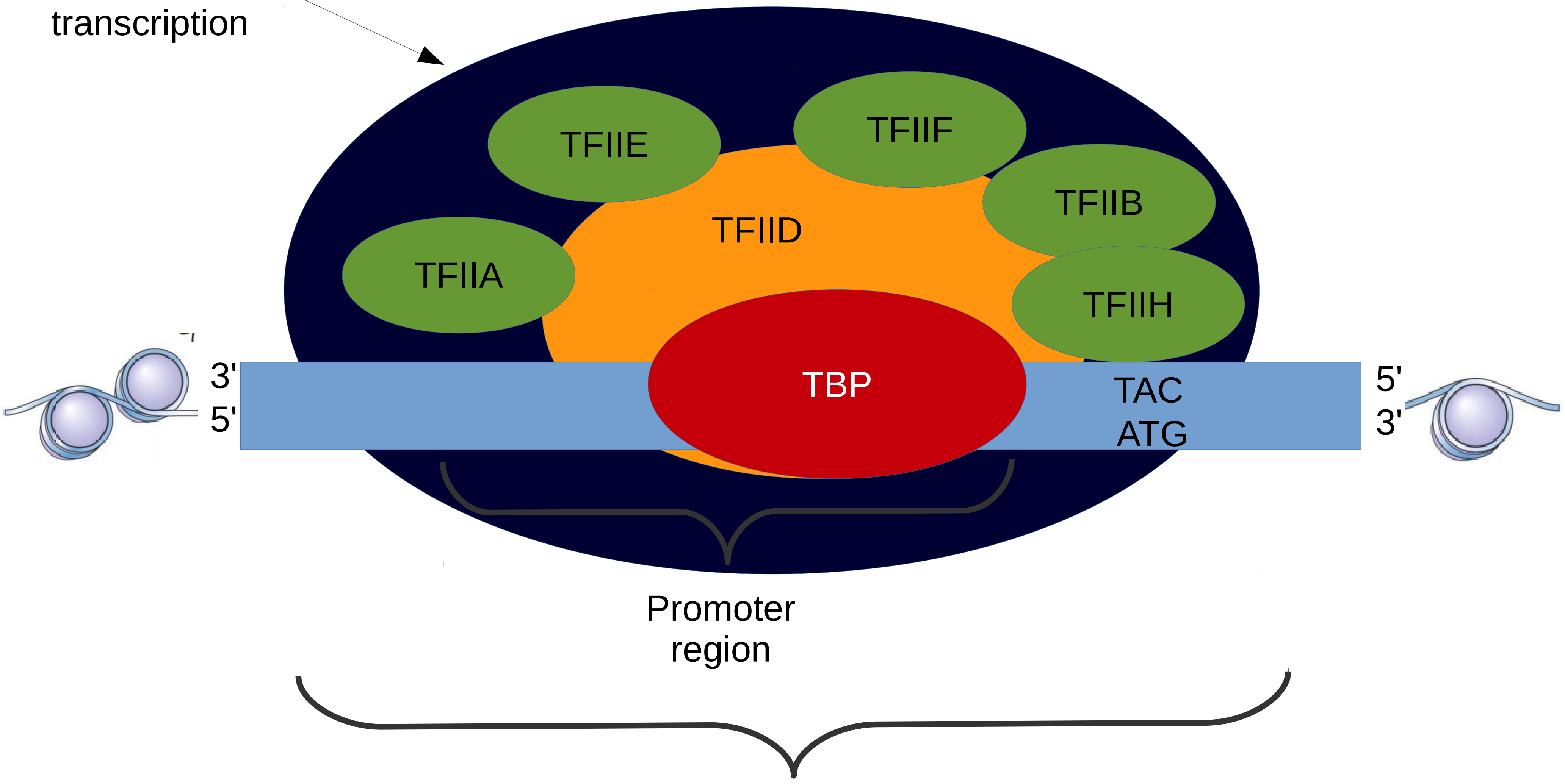


Transcription Initiation



Transcription Initiation

RNA polymerase II
Binds to the
transcription factors
and initiates
transcription

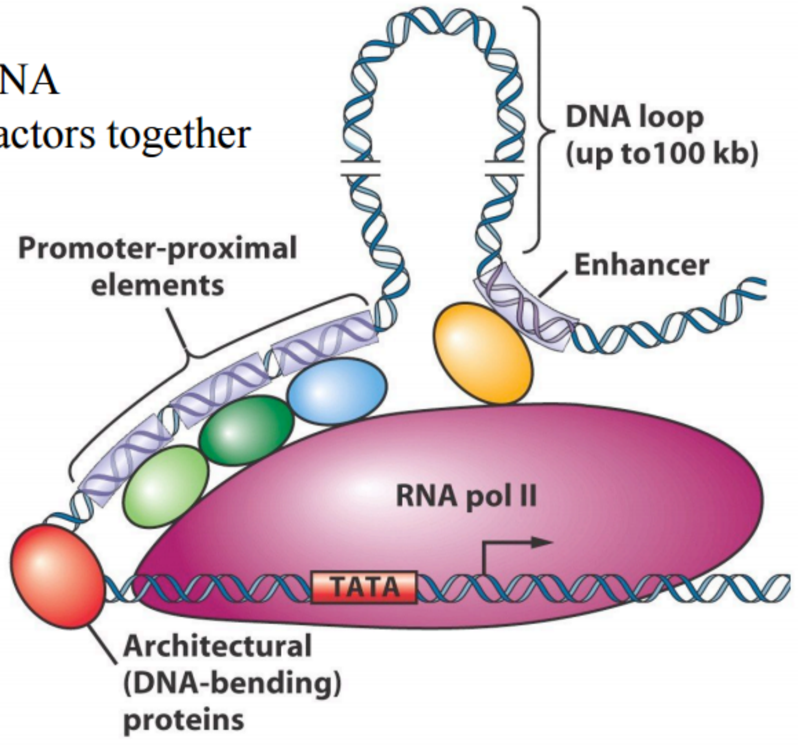


Basal factors – transcription factors that bind to the promoter region and RNA polymerase.

Enhancers

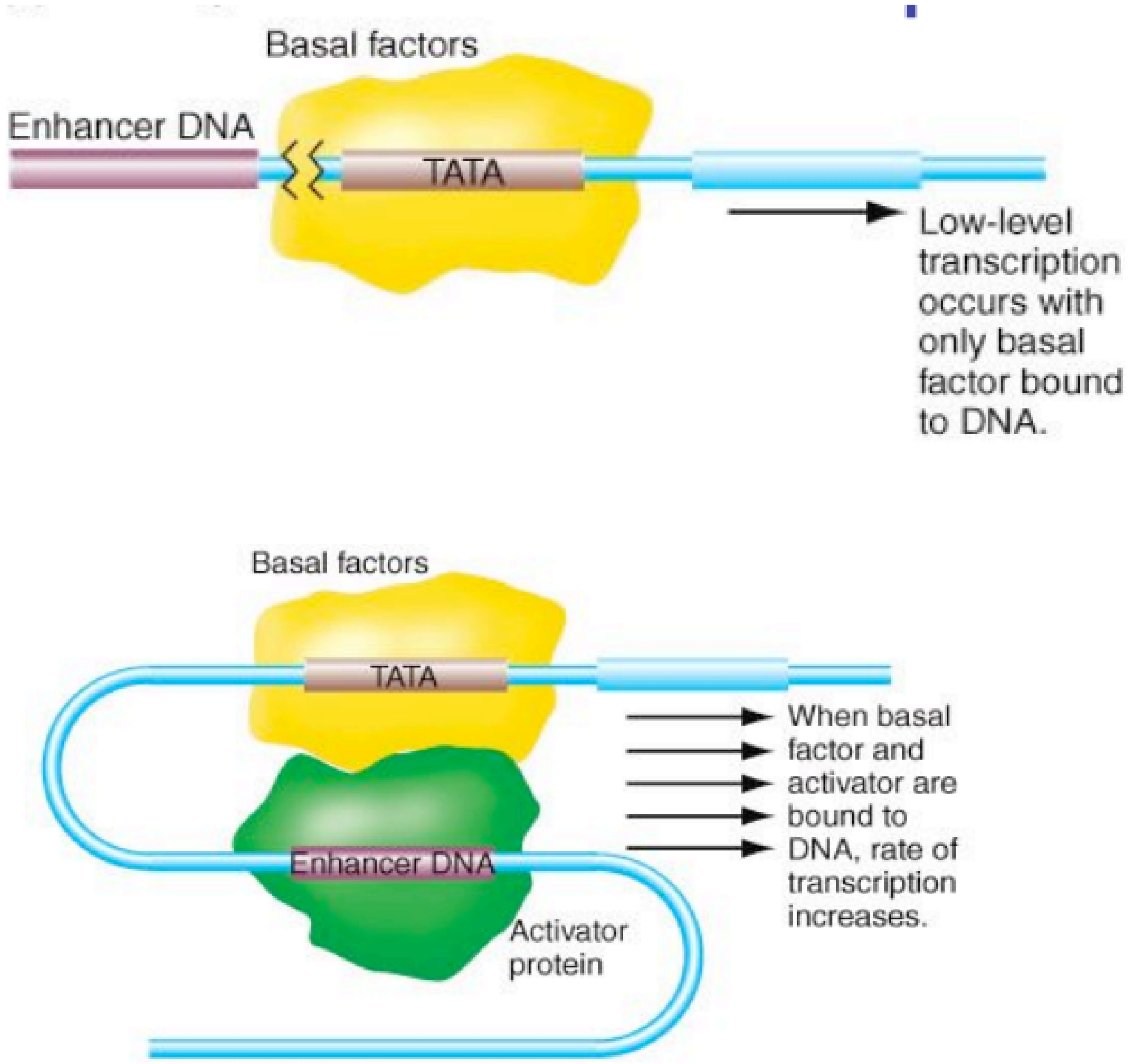
- Cis-acting element: short motif sequence elements that bind transcription factors.
- Enhancers may be located thousands of base pairs away from the transcription site.
- Enhancers help to modulate the level of protein transcription

- Possible looping of DNA
- Brings transcription factors together


DNA loop
(up to100 kb)
Promoter-proximal elements
Enhancer
RNA pol II
TATA
Architectural (DNA-bending) proteins

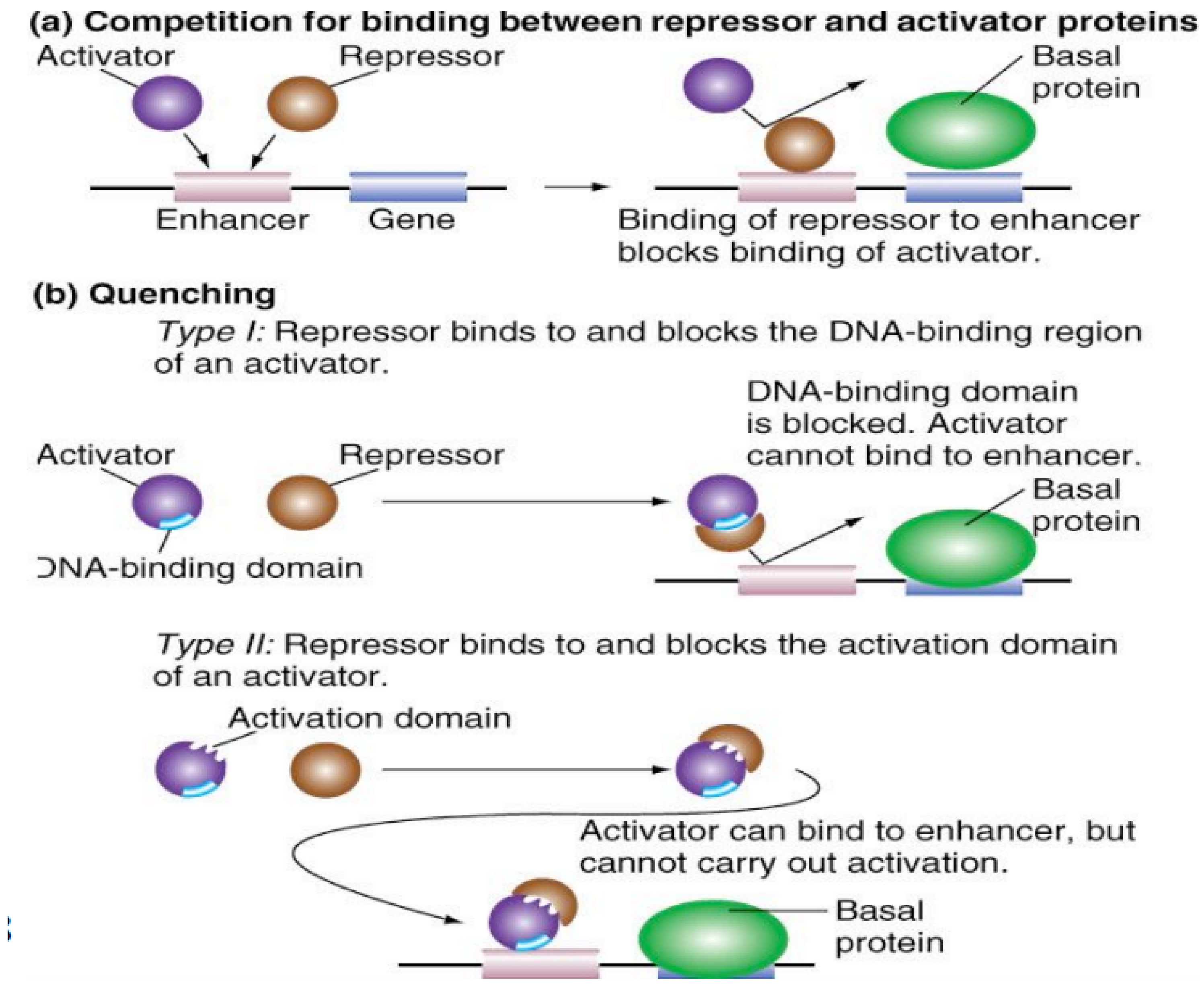
Enhancers

Activator TF's increase transcriptional activity


Basal factors
Enhancer DNA
TATA
Low-level transcription occurs with only basal factor bound to DNA.
Basal factors
TATA
Enhancer DNA
Activator protein
When basal factor and activator are bound to DNA, rate of transcription increases.

Enhancers

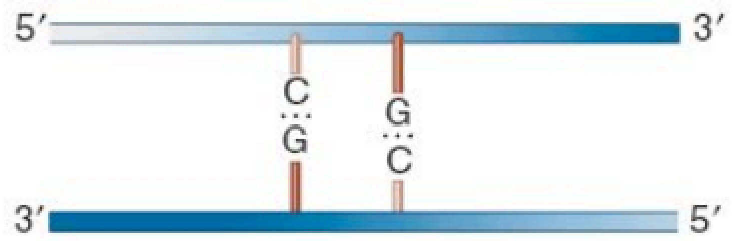
Repressor TF's diminish transcriptional activity


(a) Competition for binding between repressor and activator proteins
Activator
Repressor
Basal protein
Enhancer
Gene
Binding of repressor to enhancer blocks binding of activator.
(b) Quenching
Type I: Repressor binds to and blocks the DNA-binding region of an activator.
DNA-binding domain is blocked. Activator cannot bind to enhancer.
Activator
Repressor
Basal protein
DNA-binding domain
Type II: Repressor binds to and blocks the activation domain of an activator.
Activation domain
Activator can bind to enhancer, but cannot carry out activation.
Basal protein

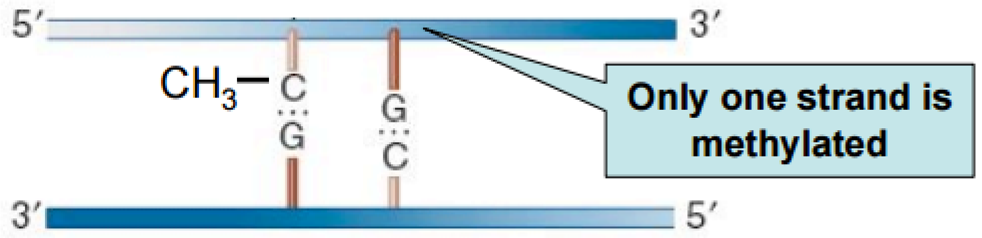
Epigenetic mechanisms of gene regulation

- Epigenetic effects refer to chemical modifications of the DNA.
  - NOT mutations! (The actual sequence of DNA does not change).

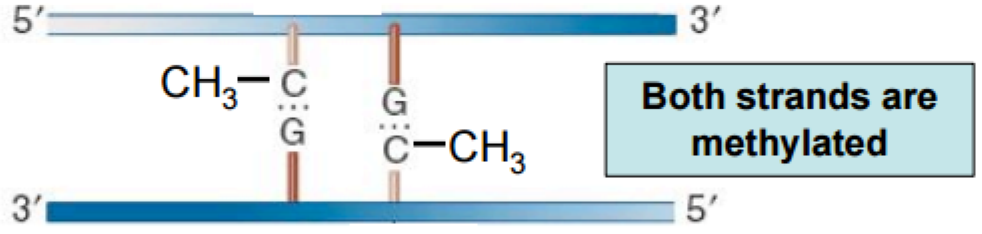
- Example: Methylation
  - DNA methylation inhibits the transcription of eukaryotic genes.
  - Methyl groups are added to cytosine in CG sequences.



un-methylated



Hemi-methylated



Fully methylated

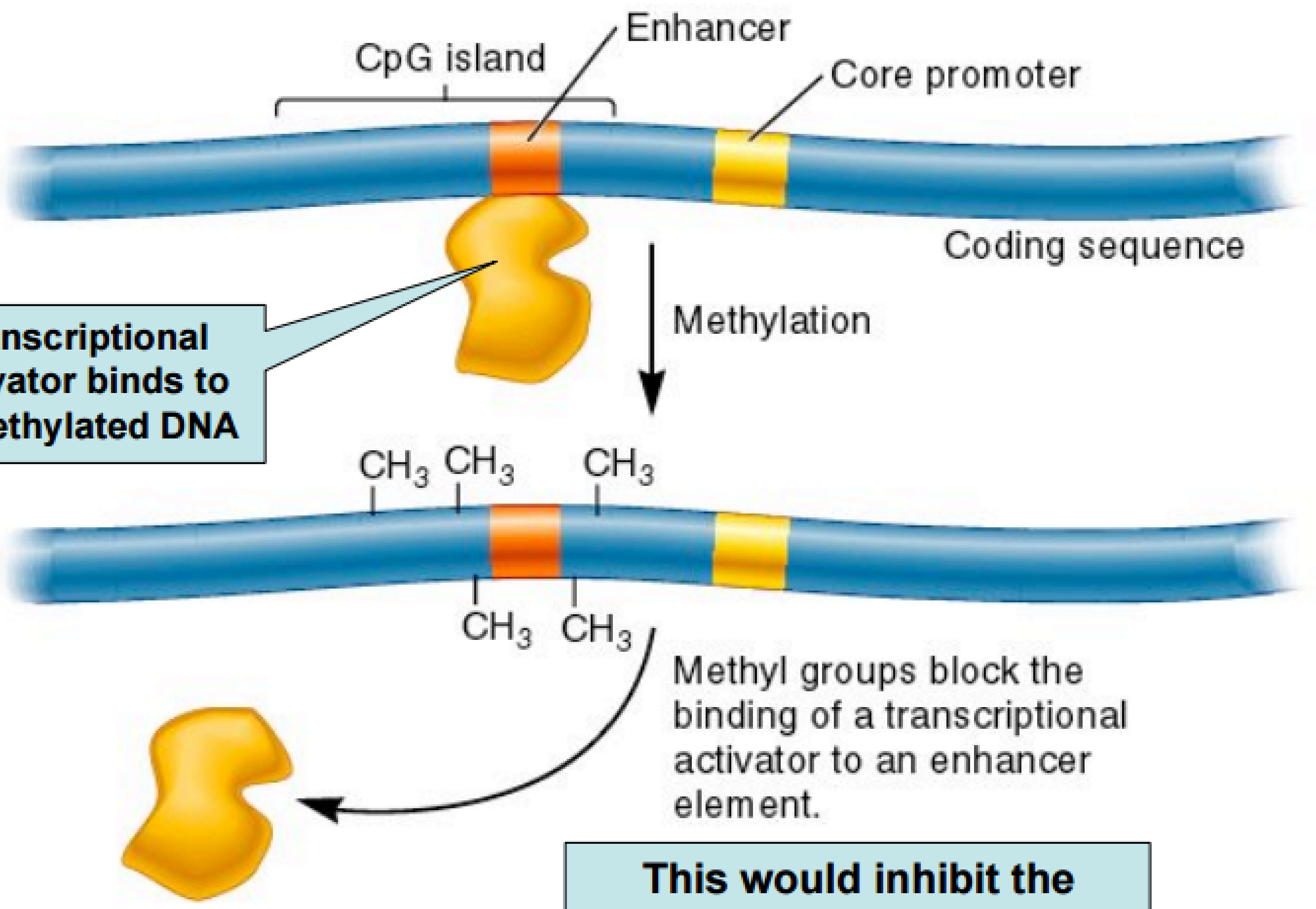
Epigenetic mechanisms of gene regulation

- Many genes have “CpG” islands near the promoter
  - 1000 – 2000 nucleotide sequences of “CG” repeats

- Housekeeping genes
  - Expressed in most cell types
  - CpG islands unmethylated

- Cell-type & tissue-type specific genes
  - Expression is silenced by methylation of CpG islands

Epigenetic mechanisms of gene regulation



Transcriptional activator binds to unmethylated DNA

This would inhibit the initiation of transcription

Part II:
Convolutional Neural Networks:
Backprop example

- Recall Calculus … **Use chain rule**

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 1$$

$$f(x,y) = \max(x,y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1(x >= y) \quad \frac{\partial f}{\partial y} = 1(y >= x)$$

Warm-up example

$$f(x,y,z) = (x+y)z$$

$$q = x + y, \;\; f = qz$$

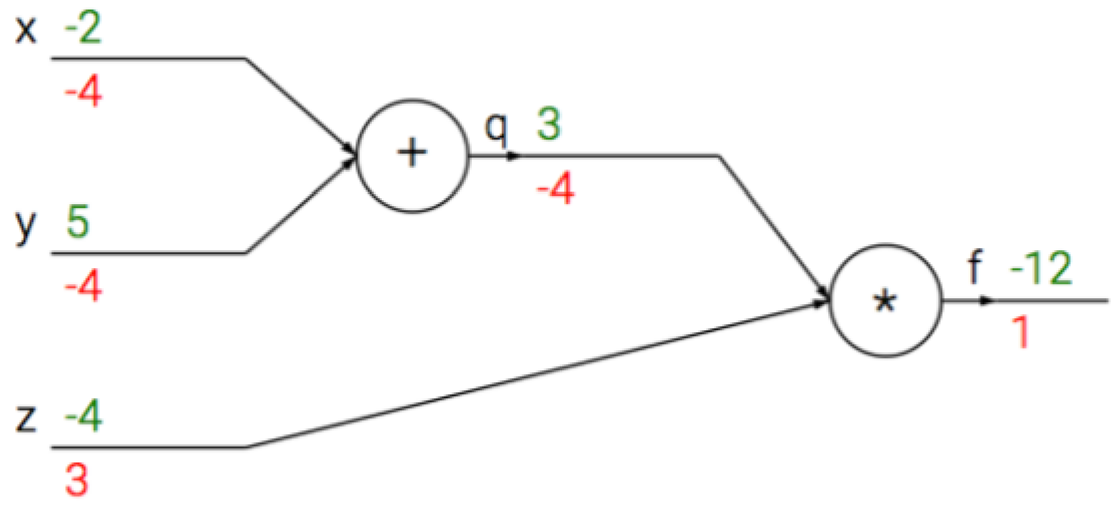
$$\Rightarrow \frac{\partial q}{\partial x} = \frac{\partial q}{\partial y} = 1$$

$$\Rightarrow \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1$$

$$\frac{\partial f}{\partial z} = q$$

```
x = -2; y = 5; z = -4
```



How would we back-propagate through a sigmoid activation?

Let's work this out on the board.

$$f(w,x)=\frac{1}{1+e^{-(w_0*x_0+w_1*x_1+w_2)}}$$

Solution:

Compute the building blocks of the sigmoid activation and find the derivative of each building block.

$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

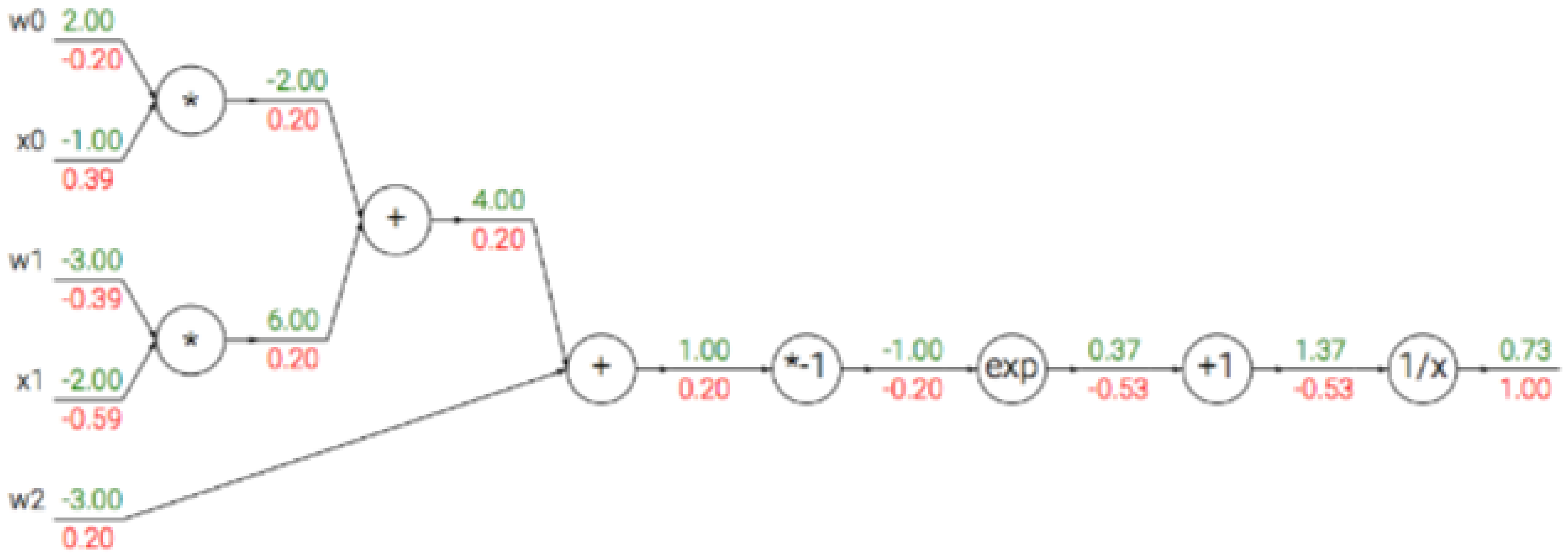
$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x = f(x)$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

Solution:

Draw a computational graph to help visualize how the building blocks connect.



Part III:
Convolutional Neural Networks:
Backprop optimizations

Batch Gradient Descent
(aka the “vanilla” gradient descent)

$$\theta = \theta - \eta * \nabla_{\theta} * J(\theta)$$

- Θ is the set of weights in the model
- η is the learning rate – this is a tunable hyperparameter, such as 0.01
- J(θ) is the value of the objective function at the particular epoch (aka. the “loss”)
- ∇θ are the gradients calculated in the backpropagation step

QUESTION: Can you think of some ways in which this weight update formula is sub-optimal if you are training on a large dataset?

Batch Gradient Descent
(aka the “vanilla” gradient descent algorithm)

$$\theta = \theta - \eta * \nabla_{\theta} * J(\theta)$$

- Θ is the set of weights in the model
- η is the learning rate – this is a tunable hyperparameter, such as 0.01
- J(θ) is the value of the objective function at the particular epoch (your loss)
- ∇θ are the gradients calculated in the backpropagation step

QUESTION: Can you think of some ways in which this weight update formula is sub-optimal if you are training on a large dataset?

ANSWER:

- You must calculate the gradients on the full dataset, which might be too large to fit into memory, and even if it does fit, the computation will be sloooow.
- You must work with a 'predetermined' dataset – no way to add new examples on the fly.

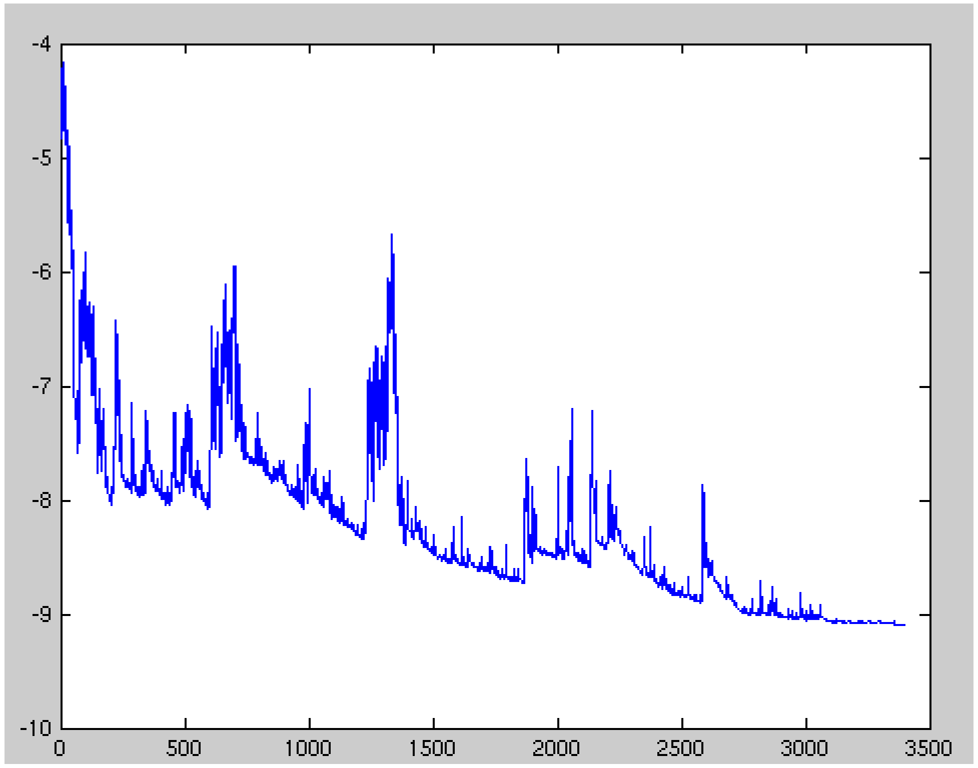
Stochastic Gradient Descent (SGD)

$$\theta = \theta - \eta * \nabla_{\theta} * J(\theta; x^{i}; y^{i})$$

- Stochastic gradient descent (SGD) in contrast performs a parameter update for **each** training example x$^{(i)}$ and label y$^{(i)}$
- This avoid redundant computations
  - In Batch gradient descent, we see many similar examples before a parameter update, so we end up computing basically the “same” gradients a bunch of times. Not true for SGD
  - Faster than “vanilla” batch gradient descent.
  - Can be used to learn “online” – you can add a new datapoint and compute the gradient in real-time.
- **Can you think of any drawbacks of using SGD?**

Stochastic Gradient Descent (SGD)

$$\theta = \theta - \eta * \nabla_\theta * J(\theta; x^i; y^i)$$



- Because you update θ for each datapoint, your loss will fluctuate a lot.
  - This can be good –the learning algorithm can “jump” to an area of lower loss without converging to a local minimum.
  - This can be bad – Once the algorithm nears the global minimum of the loss function, it will “overshoot” back and forth and have a hard time converging.

Mini-batch gradient descent: A nice intermediate between vanilla batch descent & SGD

$$\theta = \theta - \eta * \nabla_{\theta} * J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$$

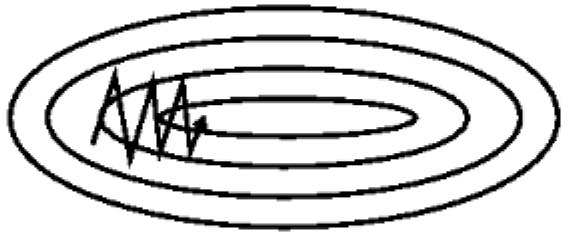
- Performs an update for every batch of n training samples.
- Reduces the variance of the parameter updates, which can lead to more stable convergence.
- Common mini-batch sizes range between 50 and 256
- Typically the algorithm of choice when training a neural network

Even with mini-batch gradient descent, challenges remain in achieving algorithm convergence.

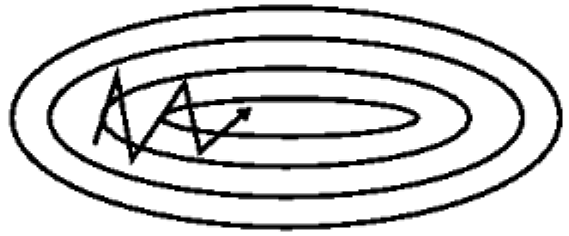
- It's hard to choose a proper learning rate.
  - A learning rate that's too low leads to slow convergence.
  - A learning rate that's too large can cause the loss function to fluctuate around the minimum or to diverge.
  - By default, the same learning rate is used for all parameters.

- The algorithm may get trapped at local minima in the loss function.

Gradient descent optimization algorithms may mitigate these challenges: **Momentum**



Without momentum



With momentum

Momentum term is set to 0.9 or less

$$v_t = \gamma * v_{t-1} + \eta * \nabla_\theta * J(\theta)$$

$$\theta = \theta - V_t$$

- Momentum adds a fraction of the update vector of the past time step to the current update vector.
- Helps SGD to avoid getting stuck in local minima of the loss function.
- Momentum helps to accelerate SGD in the relevant direction and dampens oscillation.

Gradient descent optimization algorithms may mitigate these challenges: **Nesterov accelerated gradient**

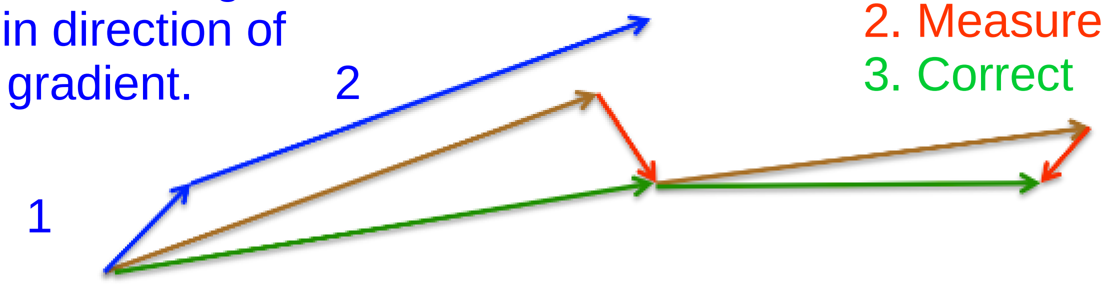
$$v_t = \gamma * v_{t-1} + \eta * \nabla_\theta * J(\theta - \gamma * v_{t-1})$$

$$\theta = \theta - V_t$$

- From the momentum approach, we know that we will compute $\theta - \gamma * V_{t-1}$ to move the parameters in the next update.
- We don't have the gradients computed yet so we cannot do the full update step.
- However, we can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters.
- NAG prevents overshooting by increasing “responsiveness” of the update step.

NAG:
1. Make a big jump in direction of previously accumulated gradient
2. Measure gradient
3. Correct

Momentum:
1. compute the current gradient
2. “big jump” in direction of accumulated gradient.



Gradient descent optimization algorithms may mitigate these challenges: **Adagrad**

- Adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.
- Uses a different learning rate for every parameter θi at every timestep t.

$$g_{t,i} = \nabla_\theta * J(\theta_i)$$

- Update rule for each parameter at timestep t:

$$\theta_{t+1,i} = \theta_{t,i} - \eta * g_{t,i}$$

- The general learning rate (η) then gets modified based on the past gradient to create our final update formula for the parameter:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} * g_{t,i}$$

- Gt here is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θi up to time step t

  - Why might this not be the best idea?

Gradient descent optimization algorithms may mitigate these challenges: **Adadelta**

- Adagrad accumulates squared gradients in the denominator
    - All added terms are positive, causing the learning rate to shrink to an infinitesimally small value over time.

- Adadelta restricts the window of accumulated past gradients to solve this problem.

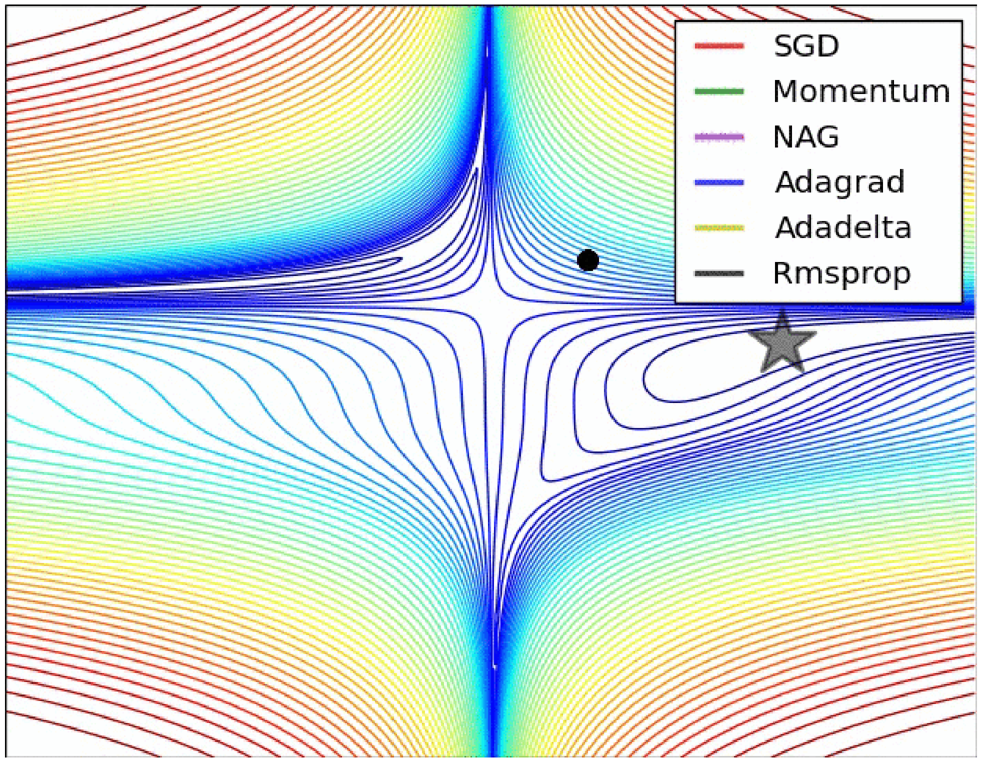
- The sum of gradients is recursively defined as a decaying average of all past squared gradients.

- The running average at a time step t depends only on the previous average and the current gradient.

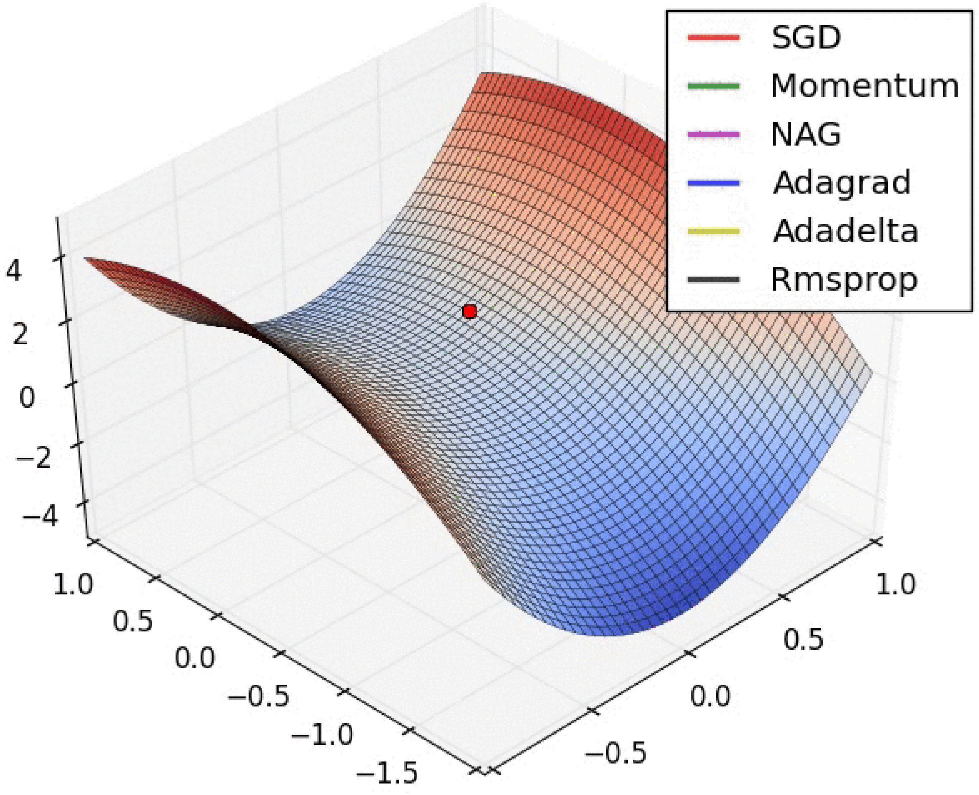
$$E[g^2]_t = \gamma * E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\Delta\theta_t = \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} * g_t$$

Optimization method head-to-head



SGD optimization on loss surface contours



SGD optimization on saddle point

Adagrad, Adadelta, RMSprop, and Adam provide the best convergence!

Part IV:
Weight initialization

All Zero Initialization

- We might guess that approximately half the weights will be positive and half will be negative. So why not initialize all weights as zeros?
- What is wrong with this approach?

All Zero Initialization

- We might guess that approximately half the weights will be positive and half will be negative. So why not initialize all weights as zeros?
- What is wrong with this approach?
- Since all weights are zero, every neuron in the network will compute the same output.
- They will then compute the same gradients during backprop.
- They will then undergo the exact same parameter updates.
- And we get nowhere :(
- We need a source of asymmetry between the neurons!

Small random numbers

- We want the weights to be close to 0, but not identically 0.
- Sample from a 0-mean, unit standard deviation gaussian.
- Great, we have symmetry breaking, but this approach is still not perfect.
- What can go wrong now? (hint: What happens to the variance of the outputs when you change the number of inputs?)

Small random numbers

- We want the weights to be close to 0, but not identically 0.
- Sample from a 0-mean, unit standard deviation gaussian, and multiply the sample by a small value, such as 0.01.
- Great, we have “symmetry breaking”, but this approach is still not perfect.
- What can go wrong now? (hint: What happens to the variance of the outputs when you change the number of inputs?)
- The distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.

Calibrating the variance with 1/sqrt(n)

- We can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs).

$$s = \sum_i^n w_i x_i$$

$$\mathrm{Var}(s) = \mathrm{Var}(\sum_i^n w_i x_i)$$

$$= \sum_i^n \mathrm{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

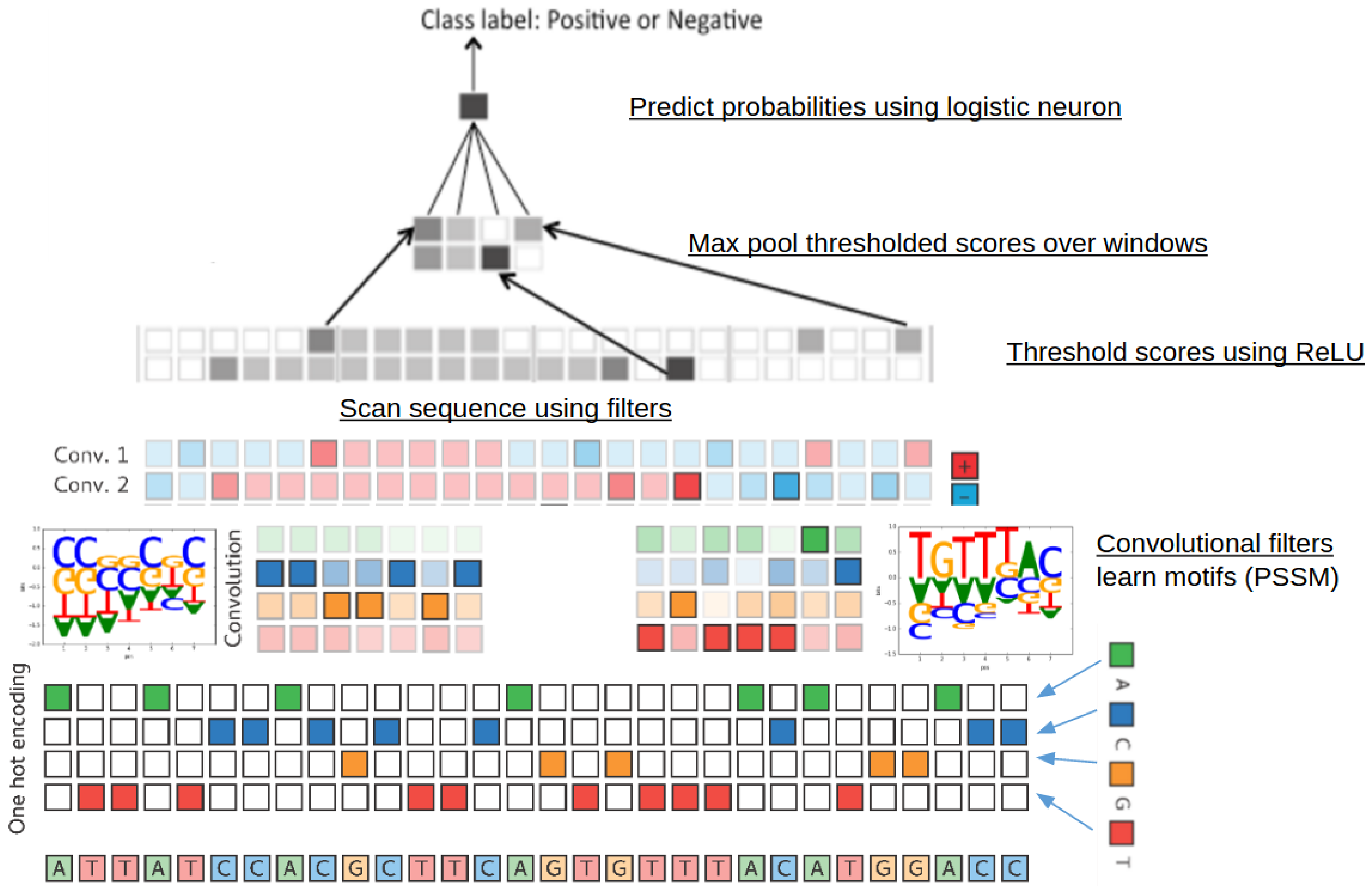
$$= (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

- In practice, we calibrate the variances by 2/sqrt(n) → Glorot initialization
- Biases are initialized to 0.

Part V:
Some helpful heuristics

A common CNN architecture template

INPUT -> [CONV -> RELU -> POOL]*n -> [FC -> RELU]*m -> FC-> Sigmoid/Softmax/Linear activation


Class label: Positive or Negative
Predict probabilities using logistic neuron
Max pool thresholded scores over windows
Threshold scores using ReLU
Scan sequence using filters
Conv. 1
Conv. 2
+
−
Convolution
Convolutional filters learn motifs (PSSM)
One hot encoding
A
C
G
T
A T T A T C C A C G C T T C A G T G T T T A C A T G G A C C

A convolutional layer….

- Accepts a volume of size $\mathbf{W_1 \times H_1 \times D_1}$
- Requires four hyperparameters:
  - Number of filters **K**,
  - their spatial extent **F**,
  - the stride **S**,
  - the amount of zero padding **P**.
- Produces a volume of size $\mathbf{W_2 \times H_2 \times D_2}$ where:
  - $\mathbf{W_2=(W_1-F+2P)/S+1}$
  - $\mathbf{H_2=(H_1-F+2P)/S+1}$ (i.e. width and height are computed equally by symmetry)
  - $\mathbf{D_2=K}$

QUESTION: With parameter sharing, how many weights will we introduce per filter?

A convolutional layer….

- Accepts a volume of size $\mathbf{W_1 \times H_1 \times D_1}$
- Requires four hyperparameters:
  - Number of filters **K**,
  - their spatial extent **F**,
  - the stride **S**,
  - the amount of zero padding **P**.
- Produces a volume of size $\mathbf{W_2 \times H_2 \times D_2}$ where:
  - $\mathbf{W_2=(W_1-F+2P)/S+1}$
  - $\mathbf{H_2=(H_1-F+2P)/S+1}$ (i.e. width and height are computed equally by symmetry)
  - $\mathbf{D_2=K}$

QUESTION: With parameter sharing, how many weights will we introduce per filter?

ANSWER: ($F*F*D_1$ weights per filter) * (K filters) =
$K*F*F*D_1$ weights and K biases

A pooling layer...

- Accepts a volume of size $\mathbf{W_1 \times H_1 \times D_1}$
- Requires two hyperparameters:
  - the spatial extent of the pool **F**,
  - the stride of the pool **S**,
- Produces a volume of size $\mathbf{W_2 \times H_2 \times D_2}$ where:
  - $\mathbf{W_2 = (W_1 - F)/S + 1}$
  - $\mathbf{H_2 = (H_1 - F)/S + 1}$
  - $\mathbf{D_2 = D_1}$