CS230: Deep Learning Midterm Review

Section 6

Agenda

1. Gradients + Propagation (Forward + Backward) (Section 2)
2. Initialization + Overfitting/Underfitting + Regularization (Section 4)
3. Optimization
4. Accuracy Metrics + Tuning
5. CNNs
6. Coding Exercise

Gradient Review

$$z = \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = x_1 y_1 + x_2 y_2 + x_3 y_3$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

$$\frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \\ \frac{\partial z}{\partial x_3} \end{bmatrix}, \frac{\partial z}{\partial y} = \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \\ \frac{\partial z}{\partial y_3} \end{bmatrix}$$

$$z = y^T x$$

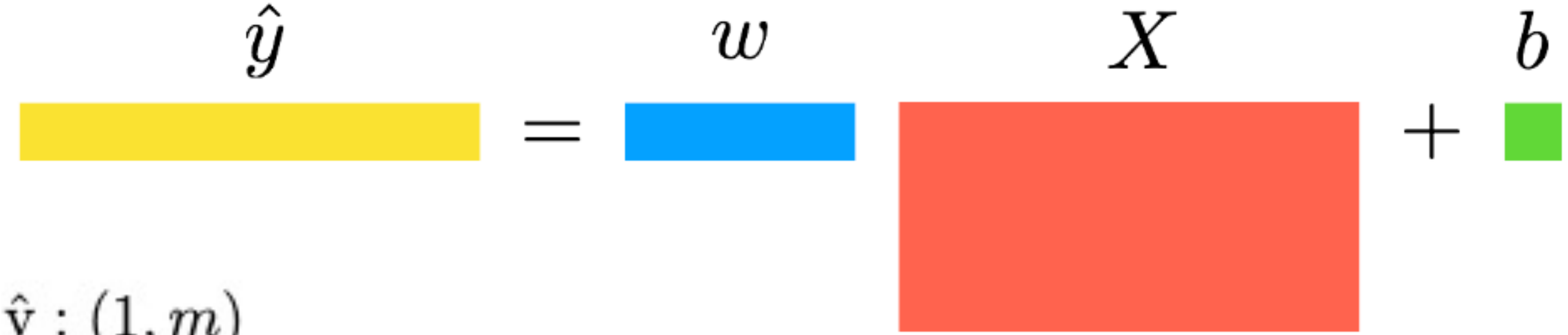
$$\frac{\partial z}{\partial x} = y$$

$$\frac{\partial z}{\partial y} = x$$

NOT THE SAME SINCE DIMENSIONS OF X AND dX MUST MATCH SINCE dX IS THE GRADIENT WITH RESPECT TO A REAL-VALUE Z

$$\frac{\partial z}{\partial x} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}, \frac{\partial z}{\partial y} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Multi-Variable Linear Regression (Forward + Batched)



$$\hat{y} : (1, m)$$
$$w : (1, F)$$
$$X : (F, m)$$
$$b : (1, 1)$$

$$\hat{y} = wX + b$$
$$L = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Note the importance of the ORDER of matrix multiplication. wX and Xw are NOT the same. Use the dimensions of input and output to see what is correct.

A matrix of size (a x b) multiplied by a matrix of size (b x c) to its right will result in a matrix of size (a x c). We need to make sure the second dimension of the left matrix is equal to the first dimension of the right matrix. In this case, both are b

Multi-Variable Linear Regression (Backward + Batched)

$$\hat{\mathrm{y}} : (1, m)$$
$$\mathrm{w} : (1, F)$$
$$\mathrm{X} : (F, m)$$
$$\mathrm{b} : (1, 1)$$

$$\hat{\mathrm{y}} = \mathrm{w}\mathrm{X} + \mathrm{b}$$
$$L = \frac{1}{m}\sum_{i=1}^{m}(\mathrm{y}_i - \hat{\mathrm{y}}_i)^2$$

$$\frac{\partial L}{\partial \hat{\mathrm{y}}_i} = \frac{\partial \frac{1}{m}(\mathrm{y}_i - \hat{\mathrm{y}}_i)^2}{\partial \hat{\mathrm{y}}_i} = \frac{2}{m}(\hat{\mathrm{y}}_i - \mathrm{y}_i)$$

$$\frac{\partial L}{\partial \hat{\mathrm{y}}} = \frac{2}{m}(\hat{\mathrm{y}} - \mathrm{y})$$

$$\frac{\partial L}{\partial \mathrm{w}} = \frac{\partial L}{\partial \hat{\mathrm{y}}}\frac{\partial \hat{\mathrm{y}}}{\partial \mathrm{w}} = \frac{\partial L}{\partial \hat{\mathrm{y}}}\mathrm{X}^T = \frac{2}{m}(\hat{\mathrm{y}} - \mathrm{y})\mathrm{X}^T$$

$$\frac{\partial L}{\partial \mathrm{X}} = \frac{\partial L}{\partial \hat{\mathrm{y}}}\frac{\partial \hat{\mathrm{y}}}{\partial \mathrm{X}} = (\frac{\partial L}{\partial \hat{\mathrm{y}}}\mathrm{w})^T = \mathrm{w}^T\frac{2}{m}(\hat{\mathrm{y}} - \mathrm{y})$$

$$\frac{\partial L}{\partial \mathrm{b}} = \frac{\partial L}{\partial \hat{\mathrm{y}}}\frac{\partial \hat{\mathrm{y}}}{\partial \mathrm{b}} = \frac{\partial L}{\partial \hat{\mathrm{y}}}\mathbf{1} = \sum_{i=1}^{m}\frac{2}{m}(\hat{\mathrm{y}}_i - \mathrm{y}_i)$$

Use the chain rule!

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)}\frac{\partial g(x)}{\partial x}$$

Gradients Exercises

- Compute the gradient of the following with respect to x,y,z

$$f(x,y,z) = x^2y + xyz + z + e^x y^3 z^2$$

- Compute the gradient of the following with respect to w_i (leave answer in terms of $\frac{\partial J}{\partial w_i}$

$$J_{L_1}(w_1,\ldots,w_l) = J(w_1,\ldots,w_l) + \lambda \sum_{i=1}^{l} |w_i|$$

,

Gradient Exercises

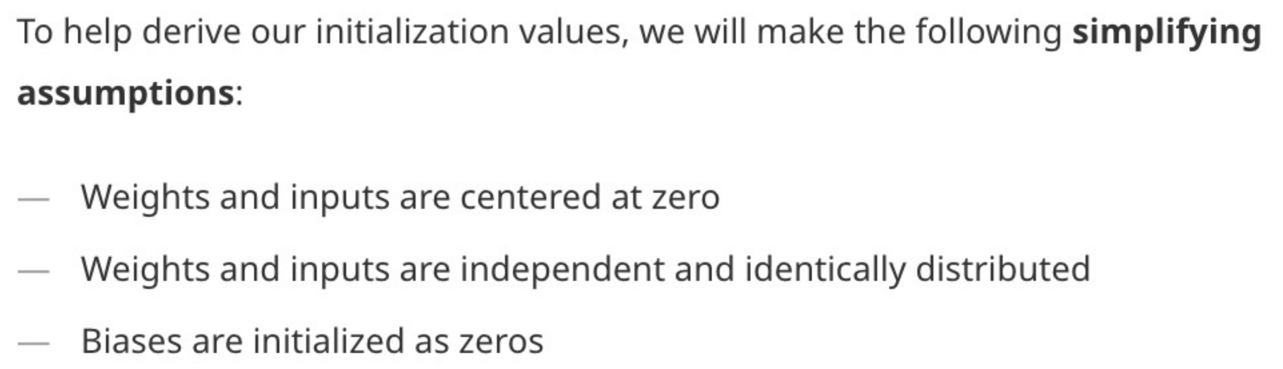
$$f(x,y,z) = x^2y + xyz + z + e^xy^3z^2$$

$$\frac{\partial f}{\partial x} = 2xy + yz + e^xy^3z^2$$
$$\frac{\partial f}{\partial y} = x^2 + xz + 3e^xy^2z^2$$
$$\frac{\partial f}{\partial z} = xy + 1 + 2e^xy^3z$$

$$\frac{\partial J_{L_1}}{\partial w_i} = \frac{\partial(J(w_1, w_2, ...w_l) + \lambda\sum_{i=1}^{l}|w_i|)}{\partial w_i}$$
$$= \frac{\partial J(w_1, w_2, ...w_l)}{\partial w_i} + \frac{\partial(\lambda\sum_{i=1}^{l}|w_i|)}{\partial w_i}$$
$$= \frac{\partial J}{\partial w_i} + \lambda\frac{\partial|w_i|}{\partial w_i} = \frac{\partial J}{\partial w_i} + \lambda\frac{\partial\begin{cases} w_i \text{ if } w_i \geq 0 \\ -w_i \text{ otherwise}\end{cases}}{\partial w_i}$$
$$= \frac{\partial J}{\partial w_i} + \lambda\begin{cases} 1 \text{ if } w_i \geq 0 \\ -1 \text{ otherwise}\end{cases} = \frac{\partial J}{\partial w_i} + \lambda\text{sign}(w_i)$$

Initialization Goals

- The general goal of initialization techniques is to initialize the weights such that the variance of the activations are the same across every layer. This constant variance helps prevent the gradient from exploding or vanishing.



To help derive our initialization values, we will make the following **simplifying assumptions**:

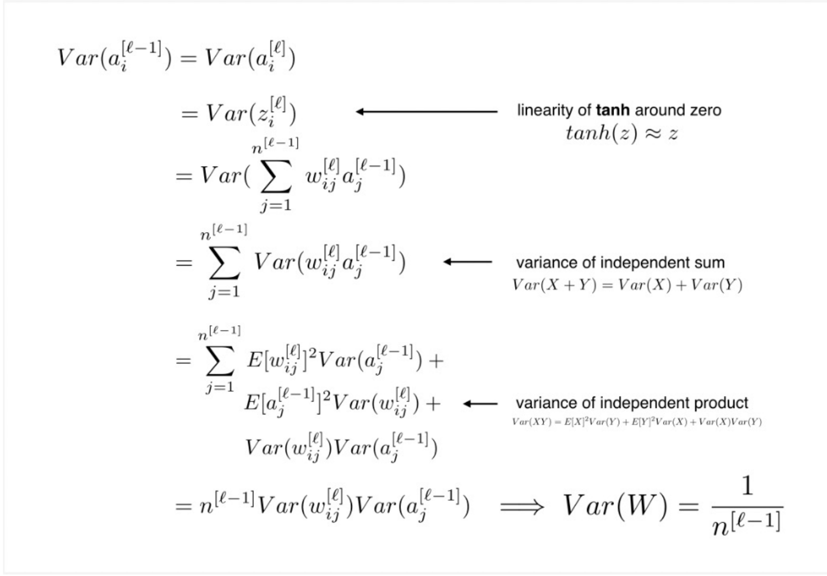
— Weights and inputs are centered at zero

— Weights and inputs are independent and identically distributed

— Biases are initialized as zeros

Xavier + He Initialization

- Xavier Initialization
  - Designed for the tanh activation function
  - Utilizes assumption that tanh is approx. linear for small inputs
  - $W_{i,j}^{[l]} = \mathcal{N}\left(0, \frac{1}{n^{[l-1]}}\right)$
- He Initialization
  - Designed for the ReLU activation function
  - Also known as Kaiming Initialization
  - Used by default by PyTorch for Linear layer



$$
\begin{aligned}
Var(a_i^{[\ell-1]}) &= Var(a_i^{[\ell]}) \\
&= Var(z_i^{[\ell]}) \longleftarrow \text{linearity of } \textbf{tanh} \text{ around zero } \; tanh(z) \approx z \\
&= Var\Big(\sum_{j=1}^{n^{[\ell-1]}} w_{ij}^{[\ell]} a_j^{[\ell-1]}\Big) \\
&= \sum_{j=1}^{n^{[\ell-1]}} Var(w_{ij}^{[\ell]} a_j^{[\ell-1]}) \longleftarrow \text{variance of independent sum } \; Var(X+Y) = Var(X) + Var(Y) \\
&= \sum_{j=1}^{n^{[\ell-1]}} E[w_{ij}^{[\ell]}]^2 Var(a_j^{[\ell-1]}) + \\
&\qquad E[a_j^{[\ell-1]}]^2 Var(w_{ij}^{[\ell]}) + \longleftarrow \text{variance of independent product } \; Var(XY) = E[X]^2Var(Y) + E[Y]^2Var(X) + Var(X)Var(Y) \\
&\qquad Var(w_{ij}^{[\ell]}) Var(a_j^{[\ell-1]}) \\
&= n^{[\ell-1]} Var(w_{ij}^{[\ell]}) Var(a_j^{[\ell-1]}) \implies Var(W) = \frac{1}{n^{[\ell-1]}}
\end{aligned}
$$

Overfitting & Underfitting

Overfitting:

- **High Variance**
- Train performance significantly better than test performance
- Validation loss increasing after a certain point

Underfitting:

- **High Bias**
- Train and test loss nearly identical
- High loss, no signs of decreasing

Combatting Overfitting & Underfitting

Overfitting:

- Increase amount of data
- Utilize data augmentation methods
- Decrease model size/complexity
- Implement early stopping
- Utilize regularization techniques

Underfitting:

- Increase model size and complexity
- Train for a longer period of time
- Reduce regularization techniques

L1 and L2 Regularization

Let's consider some cost function $J(w_1, \dots, w_l)$, a function of weight matrices $w_1, \dots, w_l$. Let's define the following two regularized cost functions:

$$J_{L_1}(w_1, \dots, w_l) = J(w_1, \dots, w_l) + \lambda \sum_{i=1}^{l} |w_i|$$

$$J_{L_2}(w_1, \dots, w_l) = J(w_1, \dots, w_l) + \lambda \sum_{i=1}^{l} ||w_i||^2$$

Let's derive the gradient updates for these cost functions.

The update for w_i when using J_{L_1} is:

$$w_i^{k+1} = w_i^k - \underbrace{\alpha\lambda sign(w_i)}_{L_1 \text{ penalty}} - \alpha \frac{\partial J}{\partial w_i}$$

The update for w_i when using J_{L_2} is:

$$w_i^{k+1} = w_i^k - \underbrace{2\alpha\lambda w_i}_{L_2 \text{ penalty}} - \alpha \frac{\partial J}{\partial w_i}$$

Define a weight matrix $\mathbf{w} \in \mathbb{R}^{m \text{ x } n}$

Denote the element in row i and column j as $\mathbf{w}_{i,j}$

$|\mathbf{w}| = \sum_{i=1}^{m} \sum_{j=1}^{n} |\mathbf{w}_{i,j}|$

$||\mathbf{w}|| = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{w}_{i,j}^2}$

$||\mathbf{w}||^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} \mathbf{w}_{i,j}^2$

$$f(x) = |x| \Rightarrow \frac{\partial f}{\partial x} = ?$$

$$f(x) = \begin{cases} x \text{ if } x \geq 0 \\ -x \text{ otherwise} \end{cases}$$

$$\frac{\partial f}{\partial x} = \begin{cases} 1 \text{ if } x \geq 0 \\ -1 \text{ otherwise} \end{cases} = \text{sign}(x)$$

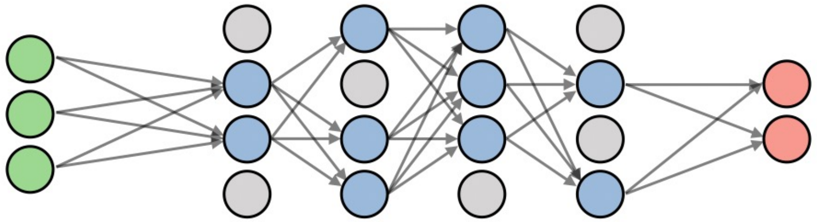
L1 Regularization generally pushes many values to 0 (sparser weights)

L2 Regularization generally more heavily impacts larger values, and doesn't force many values to 0

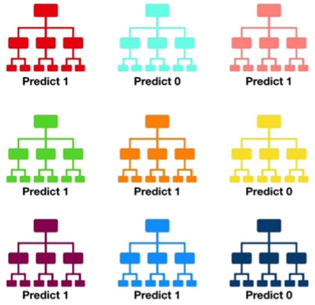
$$f(x) = x^2 \Rightarrow \frac{\partial f}{\partial x} = 2x$$

Dropout

- With some specified p, drop any given neuron of the neural network with probability p during a training iteration. Do this for each neuron independently.
- Only applies during training. During testing, all neurons are turned on and dropout is disabled.
- Helps prevent overfitting by making the model more generalizable. Doesn’t improve model’s capacity, rather focuses on generalizability of the model.

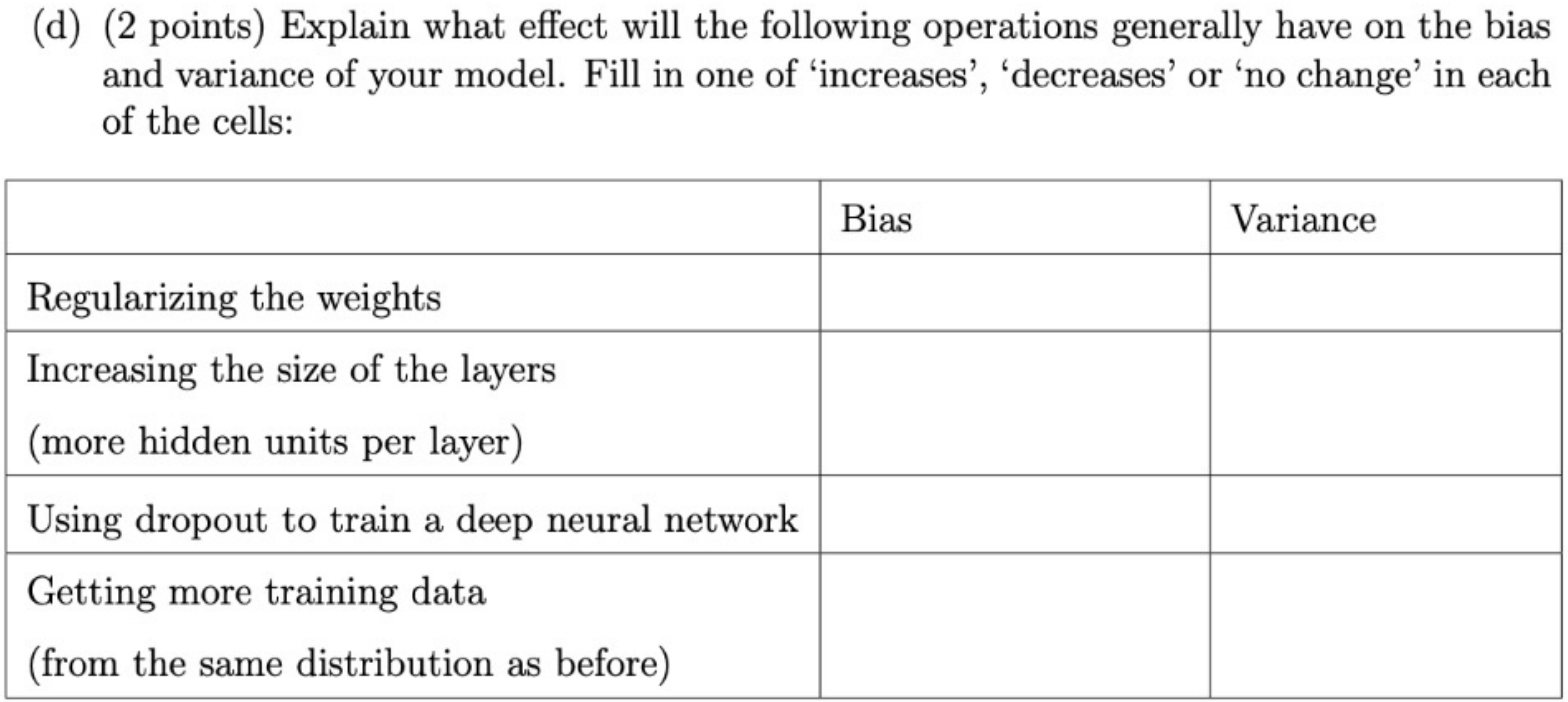


Intuitive Explanation:
We are training a forest of models to make our evaluation.
During testing, we bring all these uniquely trained models together and make an aggregate evaluation.



Avoids single neurons becoming overly important and helps equalize capacity/power of each neuron

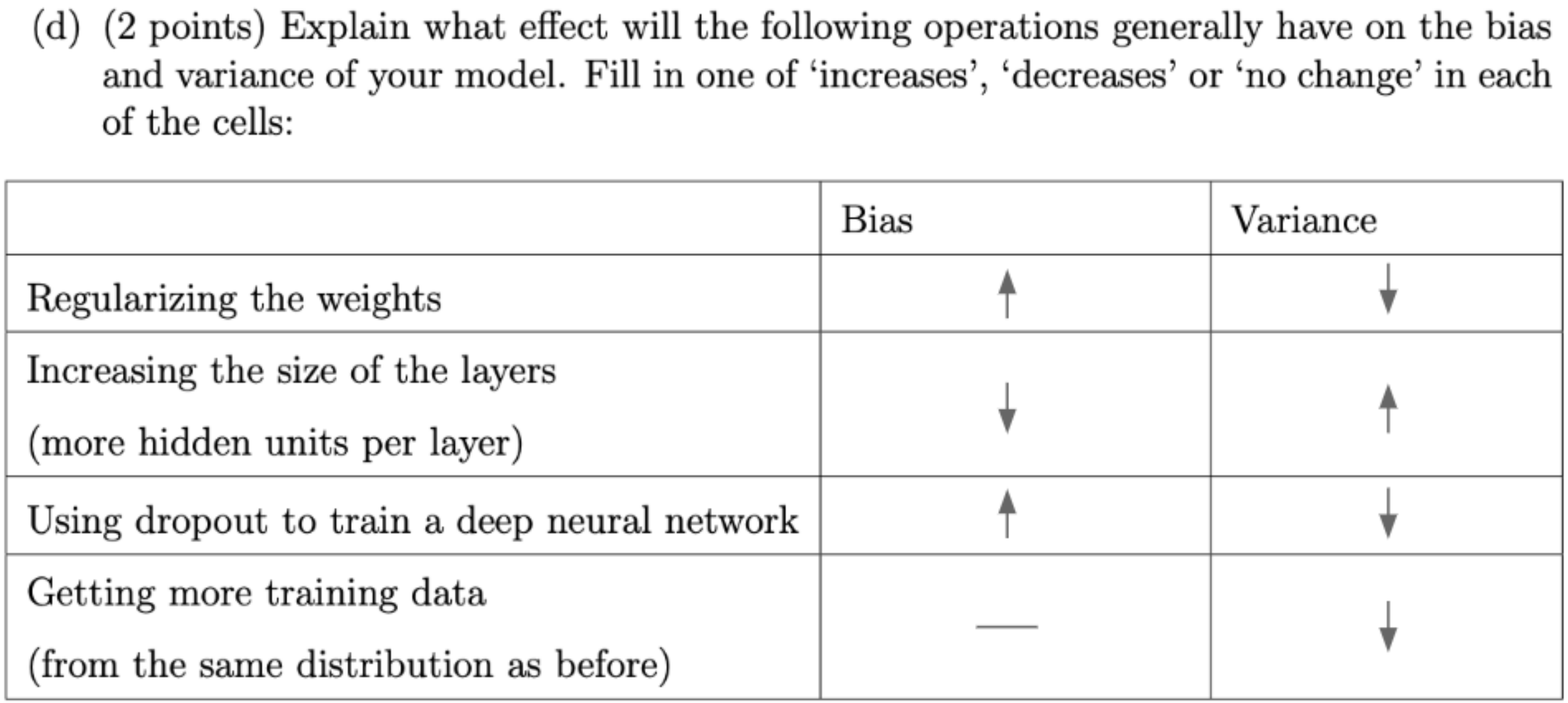
Bias/Variance Tradeoff Exercise



(d) (2 points) Explain what effect will the following operations generally have on the bias and variance of your model. Fill in one of 'increases', 'decreases' or 'no change' in each of the cells:

	Bias	Variance
Regularizing the weights		
Increasing the size of the layers (more hidden units per layer)		
Using dropout to train a deep neural network		
Getting more training data (from the same distribution as before)		

Bias/Variance Tradeoff Exercise



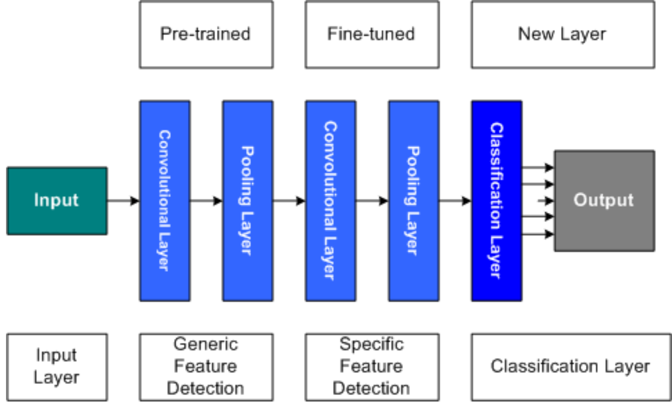
(d) (2 points) Explain what effect will the following operations generally have on the bias and variance of your model. Fill in one of 'increases', 'decreases' or 'no change' in each of the cells:

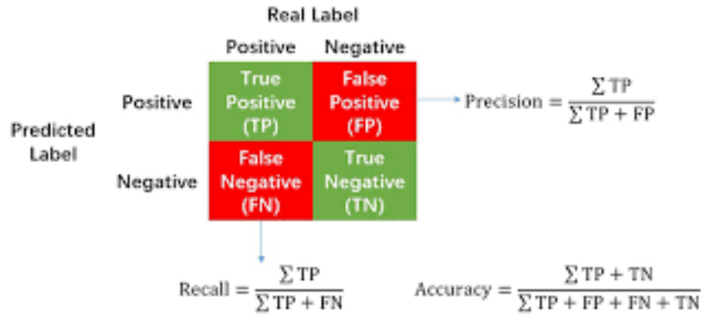
	Bias	Variance
Regularizing the weights	↑	↓
Increasing the size of the layers (more hidden units per layer)	↓	↑
Using dropout to train a deep neural network	↑	↓
Getting more training data (from the same distribution as before)	—	↓

Accuracy + Tuning

- With limited data -> utilize transfer learning.
  - Freeze early layers and fine-tune the last couple layers
  - Helps retain already learned feature mappings/weights

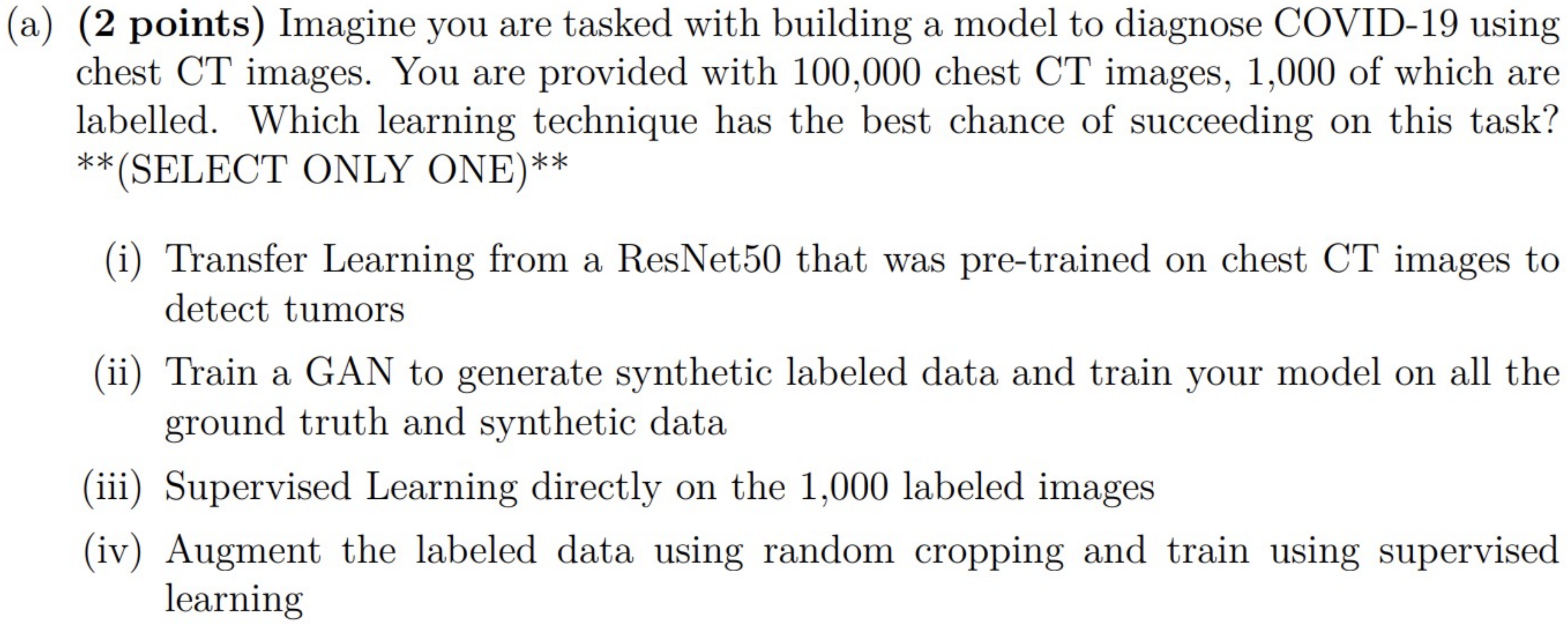
- Analyze your task for what metrics are important and prioritize optimizing those
  - Consider accuracy, precision, recall, F1 score





$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{\text{TP} + \frac{1}{2}(\text{FP} + \text{FN})}$$

Accuracy + Tuning Exercise



(a) **(2 points)** Imagine you are tasked with building a model to diagnose COVID-19 using chest CT images. You are provided with 100,000 chest CT images, 1,000 of which are labelled. Which learning technique has the best chance of succeeding on this task? **(SELECT ONLY ONE)**

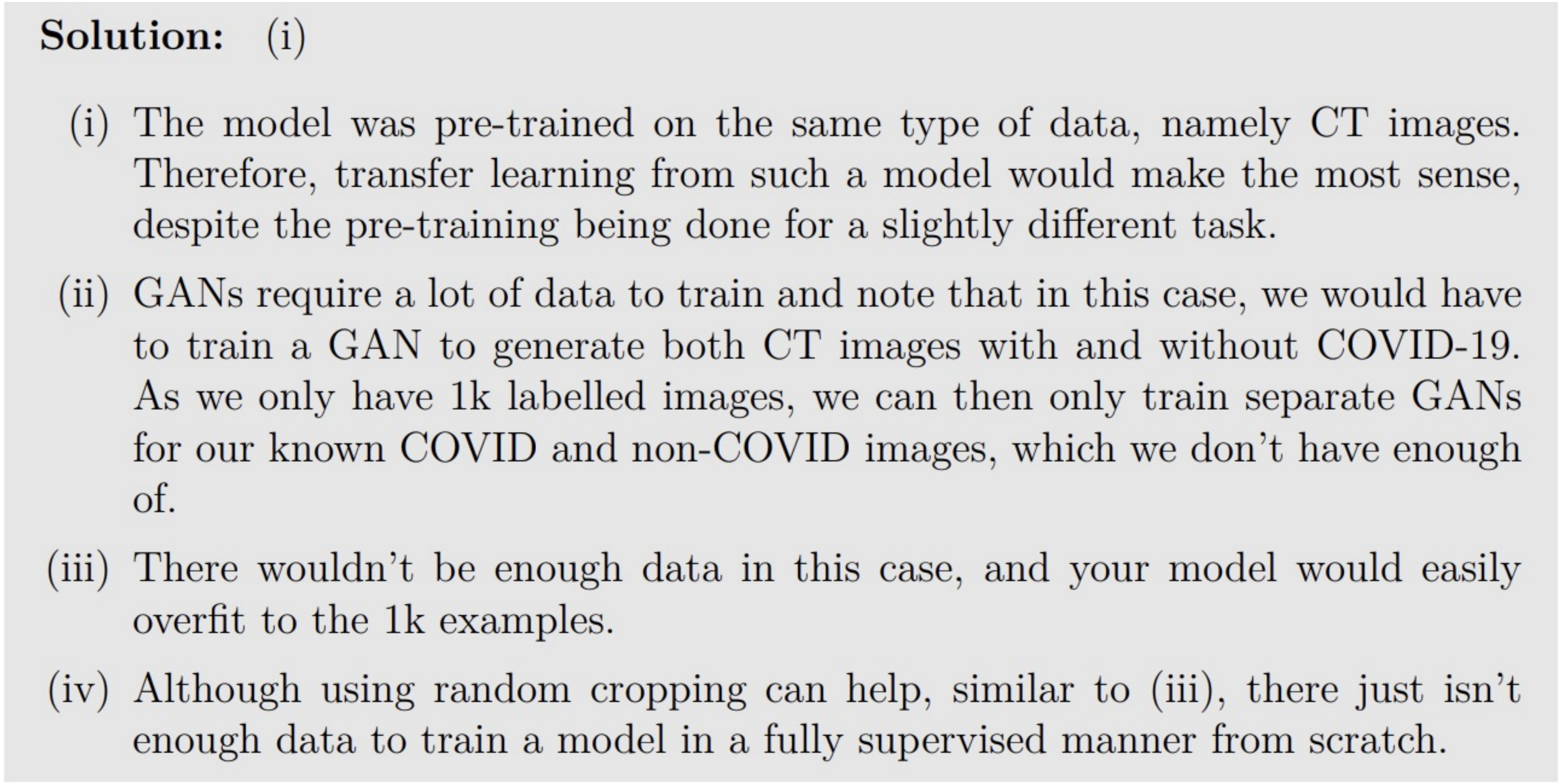
(i) Transfer Learning from a ResNet50 that was pre-trained on chest CT images to detect tumors

(ii) Train a GAN to generate synthetic labeled data and train your model on all the ground truth and synthetic data

(iii) Supervised Learning directly on the 1,000 labeled images

(iv) Augment the labeled data using random cropping and train using supervised learning

Accuracy + Tuning Exercise Answer



Solution: (i)

(i) The model was pre-trained on the same type of data, namely CT images. Therefore, transfer learning from such a model would make the most sense, despite the pre-training being done for a slightly different task.

(ii) GANs require a lot of data to train and note that in this case, we would have to train a GAN to generate both CT images with and without COVID-19. As we only have 1k labelled images, we can then only train separate GANs for our known COVID and non-COVID images, which we don't have enough of.

(iii) There wouldn't be enough data in this case, and your model would easily overfit to the 1k examples.

(iv) Although using random cropping can help, similar to (iii), there just isn't enough data to train a model in a fully supervised manner from scratch.

Accuracy + Tuning Exercise

- Your best friend is Roger Federer! He decided to come out of retirement and has made you a deal. Before each point, you get to predict if he will win or lose. You have a trained neural network to assist you, which will output p (the probability that he will win the point). Before each point, if you predict Federer will win and he does, he will reward you for your belief in him by paying you $10,000. However, if you predict that he will win the point and he loses, he will feel led on and you will have to pay him $10,000,000. If you predict he will lose the point, nothing happens**. You now have the opportunity to tune your model before the match. What accuracy metric should you be heavily optimizing for and why?**

Accuracy + Tuning Exercise Answer

You want to heavily optimize for **precision.** In this example, while correctly predicting that Federer will win the point is valuable, incorrectly predicting that he will win the point is significantly more detrimental. As such, we want to ensure that the proportion of time he is winning the point given that we predicted he wins is very high. We can optimize this with precision, which is calculated as the total true positives (we have predicted he will win the point AND he has won the point) divided by the total positive predictions (we have predicted he will win the point).

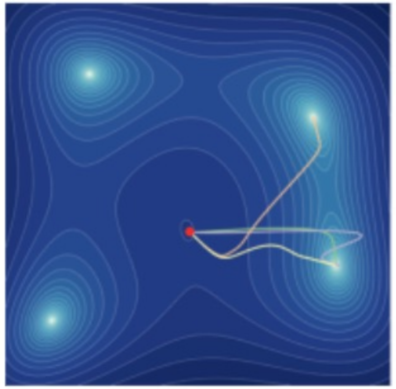
Optimization

Gradient Descent - Update parameters in the opposite direction of their gradient

Momentum - Accelerate gradient descent by adding a fraction of the update vector from previous steps to the current update

RMSProp - Adapts the learning rate to the features by dividing by the previous squared gradients.

Adam - a mixture of momentum and adaptive learning rate.



Hyperparameters

- Batch Size
- Learning Rate
- Learning Rate Decay

Optimization Exercise

Adam

Gradient Descent

Momentum

RMSProp

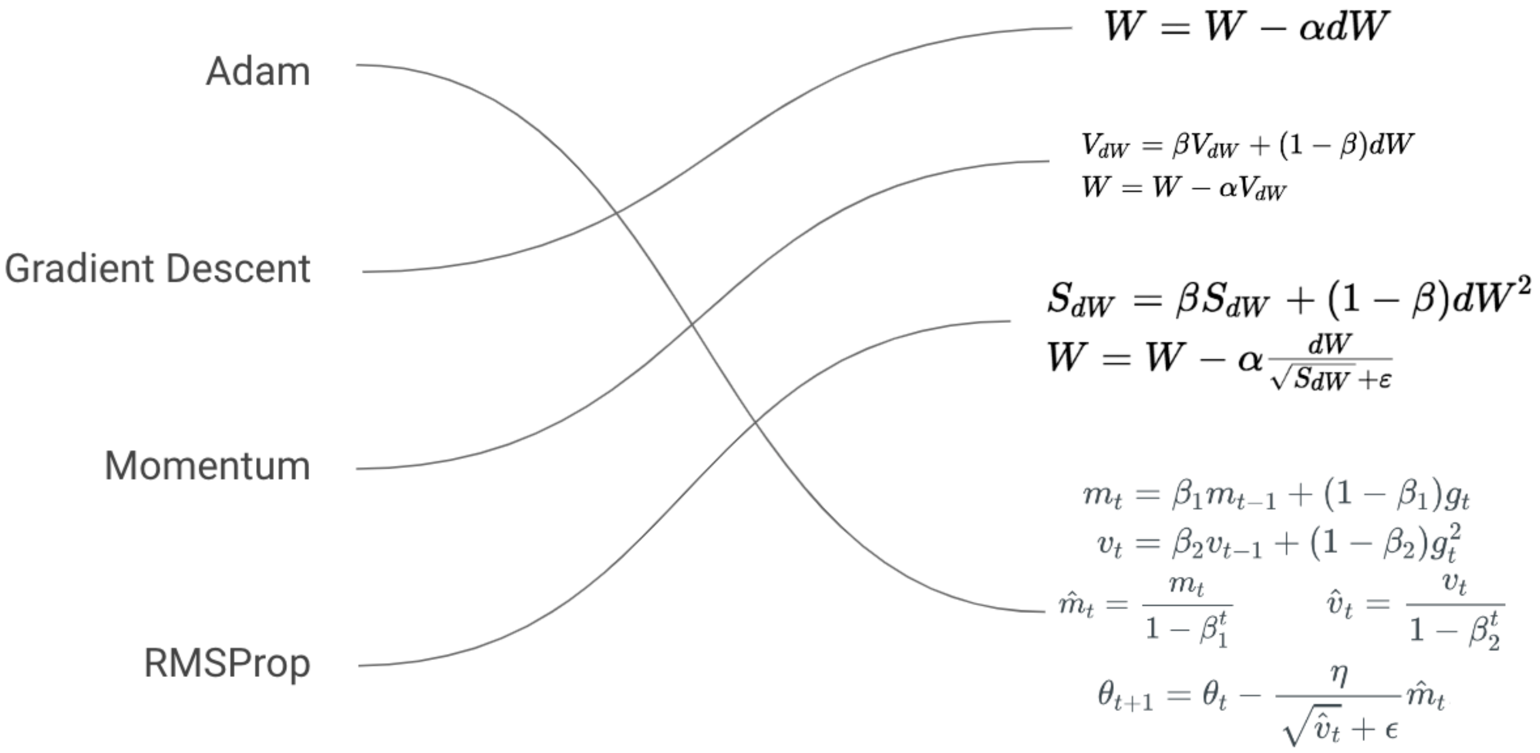
$$W = W - \alpha dW$$

$$V_{dW} = \beta V_{dW} + (1-\beta)dW$$
$$W = W - \alpha V_{dW}$$

$$S_{dW} = \beta S_{dW} + (1-\beta)dW^2$$
$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}} + \varepsilon}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

Optimization Exercise



Adam

Gradient Descent

Momentum

RMSProp

$$W = W - \alpha dW$$

$$V_{dW} = \beta V_{dW} + (1 - \beta) dW$$
$$W = W - \alpha V_{dW}$$

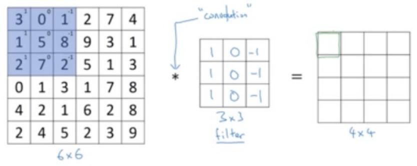
$$S_{dW} = \beta S_{dW} + (1 - \beta) dW^2$$
$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}} + \varepsilon}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

CNNs

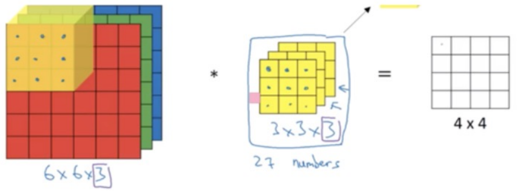
Hyperparameters: Stride (s), Padding (p), Filter size (f)

- **2D (No depth)**



# learnable parameters = (f * f (+1 for bias)) * n_f

- **3D (e.g RGB channels)**



# learnable parameters = (f * f * n_c (+1 for bias)) * n_f

- **"Valid" convolutions ⇔ No padding:**

Output shape = (n-f+1) * (n-f+1) * n_f

- **"Same" convolutions ⇔ Output shape should match Input shape**

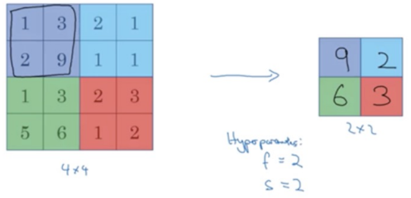
Output shape with padding = (n-f+1+2p) * (n-f+1+2p) * n_f

p for "Same" convolutions = (f-1)/2

- **General formula for output shape:** $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ **x** n_f

Pooling Layers:

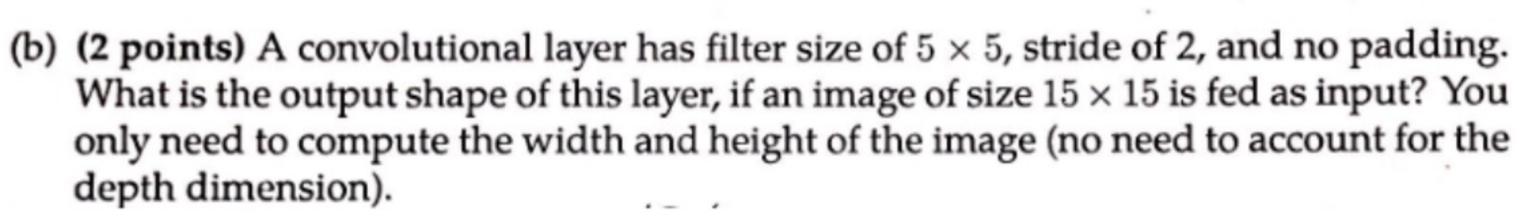
Hyperparameters: Stride (s), Filter size (f)



- **General formula for output shape:** $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$ **x** n_c

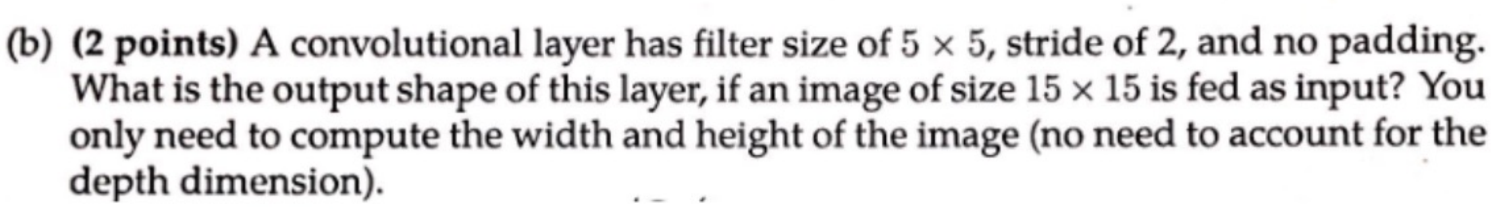
What is a convolutional layer which takes a n_c x 1 x 1 input and uses as 1 x 1 kernel (with n_f resulting channels) with stride 1?

CNN Exercise



(b) **(2 points)** A convolutional layer has filter size of 5×5, stride of 2, and no padding. What is the output shape of this layer, if an image of size 15×15 is fed as input? You only need to compute the width and height of the image (no need to account for the depth dimension).

CNN Exercise Answer


(b) **(2 points)** A convolutional layer has filter size of 5×5, stride of 2, and no padding. What is the output shape of this layer, if an image of size 15×15 is fed as input? You only need to compute the width and height of the image (no need to account for the depth dimension).

$n = 15$
$f = 5$
$s = 2$

$(15 - 5)/2 + 1 = 6$

CNN Exercise

Layer	Activation map dimensions	Number of weights	Number of biases
INPUT	$128 \times 128 \times 3$	0	0
CONV-9-32			
POOL-2			
CONV-5-64			
POOL-2			
CONV-5-64			
POOL-2			
FC-3			

CNN Exercise Answer

Layer	Activation map dimensions	Number of weights	Number of biases
INPUT	$128 \times 128 \times 3$	0	0
CONV-9-32	120 x 120 x 32	9 * 9 * 3 * 32	1 * 32
POOL-2	60 x 60 x 32	0	0
CONV-5-64	56 x 56 x 64	5 * 5 * 32 * 64	1 * 64
POOL-2	28 x 28 x 64	0	0
CONV-5-64	24 x 24 x 64	5 * 5 * 64 * 64	1 * 64
POOL-2	12 x 12 x 64	0	0
FC-3	3	3 * 12 * 12 * 64	3

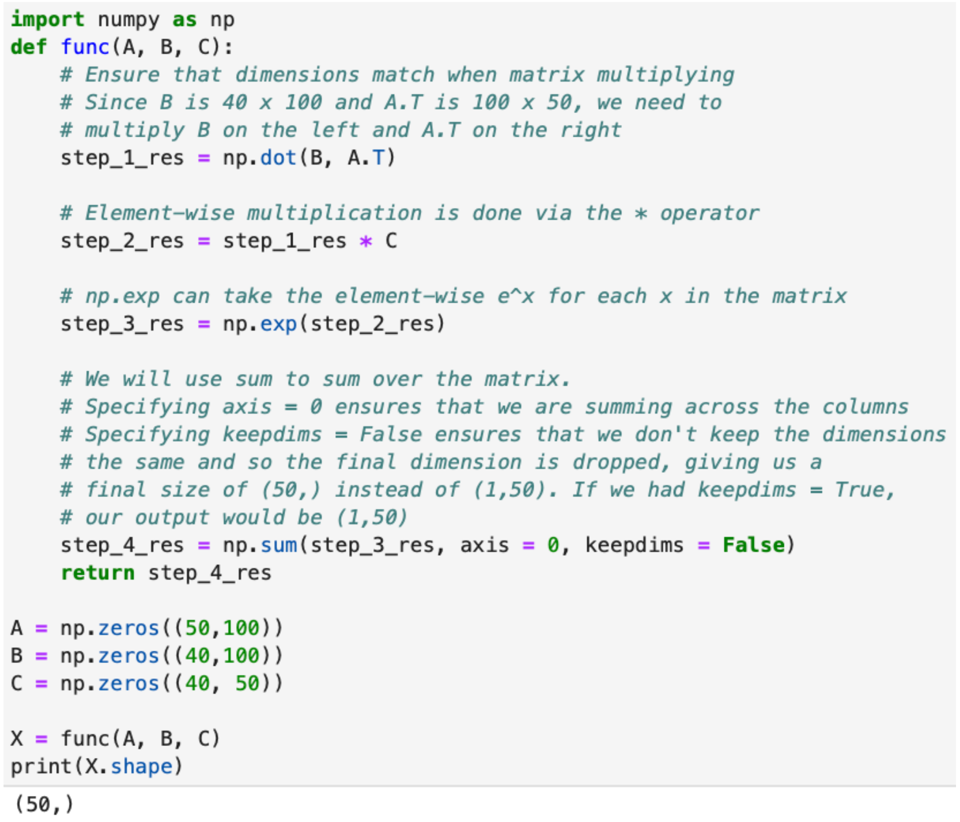
Coding Exercise

- Assume we have a function which takes in the following numpy matrices:
    - A : (50, 100), B: (40, 100), C: (40, 50)
- We would like to generate an output using the following steps:

1. Multiply A^T with B (what should the order be?)
2. Do an element-wise multiplication of the result from step 1 with C
3. Take each element in the result from 3 to the power of e (i.e., an element x is now e^x)
4. Compute the sum across each COLUMN and have our final vector be of size (50,) (notice only 1-dimensional)

Please write the code for this process

Coding Exercise Answer



```python
import numpy as np
def func(A, B, C):
    # Ensure that dimensions match when matrix multiplying
    # Since B is 40 x 100 and A.T is 100 x 50, we need to
    # multiply B on the left and A.T on the right
    step_1_res = np.dot(B, A.T)

    # Element-wise multiplication is done via the * operator
    step_2_res = step_1_res * C

    # np.exp can take the element-wise e^x for each x in the matrix
    step_3_res = np.exp(step_2_res)

    # We will use sum to sum over the matrix.
    # Specifying axis = 0 ensures that we are summing across the columns
    # Specifying keepdims = False ensures that we don't keep the dimensions
    # the same and so the final dimension is dropped, giving us a
    # final size of (50,) instead of (1,50). If we had keepdims = True,
    # our output would be (1,50)
    step_4_res = np.sum(step_3_res, axis = 0, keepdims = False)
    return step_4_res

A = np.zeros((50,100))
B = np.zeros((40,100))
C = np.zeros((40, 50))

X = func(A, B, C)
print(X.shape)
```

```
(50,)
```